THINK C

The Professional's Choice

Standard Libraries Reference

Credits

Standard Libraries Reference	Jeff Mattson
THINK C Application	Michael Kahl, Greg Howe, Mike Cote, and John McEnerney
THINK Class Library	Dan Podwall and Gregory H. Dow
Quality Assurance	David Allcott, Michael Rockhold, and Paul Vetri
Technical Support	Michael Carland, Mark Geschelin, and Phil Shapiro
Marketing Manager	Susan Smith
Product Manager	Philip Borenstein

Copyright © 1989, 1991 Symantec Corporation.
All Rights Reserved.

Symantec Corporation
10201 Torre Avenue
Cupertino, CA 95014
408/253-9600

THINK C and THINK Pascal are trademarks of Symantec Corporation. Other brands and their products are trademarks of their respective holders.

ResEdit, SARez, and SADeRez are copyrighted programs of Apple Computer, Inc. licensed to Symantec Corp. to distribute for use only in combination with THINK C. Apple software shall not be copied onto another diskette (except for archive purposes) or into memory unless as part of execution of THINK C. When THINK C has completed execution, Apple Software shall not be used by any other program.

The *Standard Libraries Reference* is copyrighted and all rights reserved. Information in this document is subject to change without notice and does not represent a commitment on the part of Symantec Corporation. The software described in this document is furnished under a license agreement. The document may not, in whole or in part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form without prior consent, in writing, from Symantec Corporation.

SYMANTEC CORPORATION MAKES NO WARRANTIES, EITHER EXPRESS OR IMPLIED, REGARDING THE ENCLOSED COMPUTER SOFTWARE PACKAGE, ITS MERCHANTABILITY, OR ITS FITNESS FOR ANY PARTICULAR PURPOSE. THE EXCLUSION OF IMPLIED WARRANTIES IS NOT PERMITTED BY SOME STATES. THE ABOVE EXCLUSION MAY NOT APPLY TO YOU. THIS WARRANTY PROVIDES YOU WITH SPECIFIC LEGAL RIGHTS. THERE MAY BE OTHER RIGHTS THAT YOU MAY HAVE WHICH VARY FROM STATE TO STATE.

Contents

Part One Getting Started

1 Welcome . . . 3
- What are the Standard Libraries? . . . 5
  - ANSI standard functions . . . 5
  - Console functions . . . 5
  - UNIX functions . . . 5
- What's in this Manual . . . 6
  - Getting Started . . . 6
  - The Standard Libraries . . . 6
  - The Standard Functions . . . 6

Part Two The Standard Libraries

2 Using the Standard Libraries . . . 9
- What Are the Standard Libraries? . . . 11
  - The ANSI libraries . . . 11
- Handling Errors . . . 11
  - File-handling errors . . . 13
- Using Files and Streams . . . 14
  - Using ANSI file functions . . . 15
  - Using UNIX file functions . . . 16
  - Naming files . . . 17
  - Binary and text files . . . 18
- Allocating Memory . . . 20
  - How the ANSI library allocates memory . . . 20
- Telling Time . . . 21
- Numerical Limits and Parameters . . . 21
  - Floating-point parameters and limits . . . 22
  - Integer limits . . . 22
- The Standard Library Functions . . . 24
  - Error functions . . . 26
  - I/O functions . . . 26
  - Math functions . . . 26
  - Character functions . . . 29
  - String functions . . . 30
  - Time functions . . . 30
  - Memory allocation functions . . . 32
    - . . . 33

Exiting functions . . . 33
Searching and sorting functions . . . 33
String conversion functions . . . 33
Variable argument list functions . . . 34
Jumping functions . . . 34
Signal-handling functions . . . 34
UNIX functions . . . 34
Multibyte and wide character functions . . . 35
Locale functions . . . 36
OS functions . . . 36

3 Using Console Windows . . . 37
What Is the Console Package? . . . 39
Using the Console Package . . . 40
Pausing a program when it's done . . . 41
Handling Command-. (Command-Period) . . . 41
Using your windows in the console interface . . . 43
Using consoles outside the console interface . . . 44
Modifying Console Windows . . . 44
Setting console options . . . 44
Setting the input mode . . . 45
Using inverse video . . . 46

4 Using printf and scanf . . . 49
Using Formatted I/O . . . 51
Using formatted output . . . 51
Using formatted input . . . 52
Printing Integers . . . 55
Printing plus signs . . . 56
Printing pointers . . . 57
Printing alternate forms of hexadecimal and octal integers . . . 57
Specifying the number of digits to print . . . 58
Specifying the total width of the number . . . 58
Reading Integers . . . 60
Specifying the field size . . . 62
Letting the function decide the base . . . 62
Printing Floating-Point Numbers . . . 63
Printing in floating-point notation . . . 64
Printing in exponential notation . . . 65
Letting the function choose the notation . . . 65
Printing plus signs . . . 67
Specifying the total width of the number . . . 67
Printing Infinity and NaNs . . . 68
Reading Floating-Point Numbers . . . 69
Specifying the field size . . . 70
Printing Strings and Characters . . . 71
Printing Strings . . . 71
Printing characters . . . 73
Reading Strings and Characters . . . 73
Reading strings . . . 73
Reading characters . . . 75

Formatting Macintosh Toolbox Types . . . 75
Using OSType and ResType (four-byte constants) . . . 75
Using Pascal strings . . . 76
Advanced Features . . . 78
Getting the precision and field size from the argument list . . . 78
Suppressing assignment . . . 79
Finding the number of characters read or written . . . 80
Using scan sets . . . 82
printf Summary . . . 83
Flag character . . . 84
Field size . . . 85
Precision directive . . . 85
Argument size . . . 86
Conversion characters . . . 87
scanf Summary . . . 89
Flag character . . . 90
Field size . . . 90
Argument size . . . 90
Conversion characters . . . 91

Part Three The Standard Functions

5 The Standard Library Functions . . . 95
Using the Function Descriptions . . . 99
Reading the descriptions . . . 99
Using functions not in the ANSI standard . . . 99
`abort()` . . . 101
`abs()` . . . 102
`acos()` . . . 103
`asctime()` . . . 104
`asin()` . . . 105
`assert()` . . . 106
`atan()` . . . 107
`atan2()` . . . 108
`atexit()` . . . 109
`_atexit()` . . . 111
`atof()` . . . 113
`atoi()` . . . 114
`atol()` . . . 115
`bsearch()` . . . 116
`calloc()` . . . 119
`ccleol()` . . . 120
`ccleos()` . . . 121
`ccommand()` . . . 122
`cecho2file()` . . . 125
`cecho2printer()` . . . 126
`ceil()` . . . 128
`cgetxy()` . . . 129
`cgotoxy()` . . . 130
`chide()` . . . 131
`cinverse()` . . . 133

clearerr() . . . 134
clock() . . . 135
close() . . . 136
cos() . . . 137
cosh() . . . 138
creat() . . . 139
csetmode() . . . 140
csettabs() . . . 142
cshow() . . . 143
ctime() . . . 145
difftime() . . . 146
div() . . . 147
exec(), execl(), execle(), execlp(), execlpe(), execv(), execve(), execvp(), execvpe() . . . 148
exit() . . . 149
_exit() . . . 150
_exiting() . . . 152
exp() . . . 154
fabs() . . . 155
fclose() . . . 156
fdopen() . . . 157
feof() . . . 158
ferror() . . . 159
fflush() . . . 160
fgetc() . . . 162
fgetpos() . . . 164
fgets() . . . 166
fileno() . . . 168
floor() . . . 169
fmod() . . . 170
fopen() . . . 171
fopenc() . . . 174
fprintf() . . . 175
fputc() . . . 177
fputs() . . . 178
fread() . . . 179
free() . . . 181
freopen() . . . 182
freopenc() . . . 184
frexp() . . . 185
fscanf() . . . 187
fseek() . . . 189
fsetpos() . . . 191
ftell() . . . 193
fwrite() . . . 195
getc() . . . 197
getchar() . . . 199
getegid(), geteuid(), getgid(), getpgrp(), getpid(), getppid(), getuid() . . . 201
getenv() . . . 202
getlogin() . . . 203
gets() . . . 204

getw() . . . 205
gmtime() . . . 206
isatty() . . . 207
isxxxx() . . . 208
kill() . . . 210
labs() . . . 211
ldexp() . . . 212
ldiv() . . . 213
localeconv() . . . 214
localtime() . . . 215
log() . . . 216
log10() . . . 217
lseek() . . . 218
longjmp() . . . 220
malloc() . . . 222
mblen() . . . 223
mbstowcs() . . . 224
mbtowc() . . . 225
memchr() . . . 226
memcmp() . . . 227
memcpy() . . . 228
memmove() . . . 229
memset() . . . 230
mktime() . . . 231
modf() . . . 233
open() . . . 234
perror() . . . 236
pow() . . . 237
printf() . . . 238
putc() . . . 239
putchar() . . . 241
puts() . . . 242
putw() . . . 243
qsort() . . . 244
raise() . . . 246
rand() . . . 248
read() . . . 249
realloc() . . . 251
remove() . . . 253
rename() . . . 254
rewind() . . . 255
scanf() . . . 256
setbuf() . . . 258
setjmp() . . . 260
setlocale() . . . 262
setvbuf() . . . 263
signal() . . . 265
sin() . . . 268
sinh() . . . 269
sleep() . . . 270
sprintf() . . . 271
sqrt() . . . 272

srand() . . . 273
sscanf() . . . 274
strcat() . . . 275
strchr() . . . 276
strcmp() . . . 277
strcoll() . . . 278
strcpy() . . . 279
strcspn() . . . 280
strerror() . . . 281
strftime() . . . 282
strlen() . . . 284
strncat() . . . 285
strncmp() . . . 287
strncpy() . . . 289
strpbrk() . . . 290
strrchr() . . . 291
strspn() . . . 292
strstr() . . . 293
strtod() . . . 294
strtok() . . . 296
strtol() . . . 298
strtoul() . . . 300
strxfrm() . . . 302
system() . . . 303
tan() . . . 304
tanh() . . . 305
tell() . . . 306
time() . . . 308
tmpfile() . . . 309
tmpnam() . . . 310
tolower() . . . 311
toupper() . . . 313
ungetc() . . . 315
unlink() . . . 317
va_start(), va_arg(), va_end() . . . 318
vfprintf() . . . 321
vprintf() . . . 323
vsprintf() . . . 325
wctomb() . . . 327
wcstombs() . . . 328
write() . . . 329

Index . . . 331

THINK C
Getting Started

Part One
1 Welcome

Welcome 1

Welcome to the *Standard Libraries Reference*. This manual describes the standard libraries. The standard libraries are unix, `ANSI`, `ANSI-A4`, and `ANSI-small`.

Contents

What are the Standard Libraries? 5
- ANSI standard functions 5
- Console functions 5
- UNIX functions 6

What's in this Manual 6
- Getting Started 6
- The Standard Libraries 6
- The Standard Functions 6

What are the Standard Libraries?

THINK C's standard libraries provide you with functions for such common operations as file and screen I/O, string-handling, and math. There are three types of functions in the standard libraries:

- ANSI Standard Functions
- Console Functions
- UNIX Functions.

ANSI standard functions

The ANSI standard functions are utility functions that let you perform simple I/O, string-handling, math, and more. They form a complete implementation of the library defined in the ANSI standard for C.

The document describing the ANSI standard is *American National Standard for Information Systems—Programming Language—C X3.159-1989.* Cost is $50, plus $5 for shipping and handling. To order a copy, write or call:

American National Standards Institute (ANSI)
Sales Dept
1430 Broadway
New York, NY 10018
(212) 642-4900

Most of Chapter 2, "Using the Standard Libraries," describes how to use these functions. Each function is described in detail in Chapter 5, "The Standard Library Functions." Chapter 4, "Using printf and scanf," tells you how to form your input and output with functions like `scanf()` and `printf()`.

Console functions

The console functions let you create and manipulate consoles, simple Macintosh windows that act like the displays of MS-DOS or UNIX computers. Consoles let you create and port programs without a Macintosh interface.

The console functions are part of the console package. If your program uses a standard I/O stream (`stdin`, `stdout`, or `stderr`), the console package creates a console to print to and read from, and gives your program a simple interface. Use the console functions if you need to manipulate consoles, or if you want to use consoles in a program with your own interface.

These functions are in the THINK C `ANSI` library and are an extension to the ANSI standard library. Chapter 3, "Using Console Windows," describes how to use console windows. Each console function is described in detail in Chapter 5, "The Standard Library Functions."

UNIX functions

The UNIX functions are a collection of functions commonly found on UNIX systems that are not in the ANSI standard library. They help you port UNIX programs quickly. To ensure portability, don't use these functions in new programs. Instead, use the equivalent functions in the ANSI library or the Macintosh Toolbox.

Much of Chapter 2, "Using the Standard Libraries," describes how to use these functions, especially sections "Using UNIX file functions" on page 17, and "UNIX functions" on page 34. Each function is described in detail in Chapter 5, "The Standard Library Functions."

What's in this Manual

This manual is organized in three sections: Getting Started, The Standard Libraries, and The Standard Functions.

Getting Started

This is the section you're reading. It contains this chapter.

The Standard Libraries

This chapter contains three chapters that describe how to use THINK C's standard libraries:

The chapter...	describes...
Using the Standard Libraries	How to use the `ANSI` and `unix` libraries.
Using Console Windows	Consoles, which are Macintosh windows that act like dumb terminals, and the functions that manipulate them.
Using printf and scanf	How to format text with `printf()`, `scanf()`, and related functions.

The Standard Functions

This chapter contains one chapter that describes the functions in the THINK C standard libraries:

The chapter...	describes...
The Standard Library Functions	Each function in the `ANSI` and `unix` libraries.

THINK C

The Standard Libraries

Part Two

2 Using the Standard Libraries
3 Using Console Windows
4 Using printf and scanf

Using the Standard Libraries 2

This chapter gives an overview of the THINK C standard libraries: `ANSI` and `unix`. It touches on the following topics, pointing out areas that may confuse someone familiar with a UNIX or MS-DOS implementation of the library:

- Handling errors
- Using files and streams
- Allocating memory
- Telling time
- Numerical limits and parameters

It also includes a list of all the functions in the ANSI library organized by category. For a detailed description of each function, see Chapter 5, "The Standard Library Functions."

What you should know

You should be familiar with C and the ANSI standard library. For more detailed information, see a good C tutorial, such as *The C Programming Language, Second Edition* by Kernighan and Ritchie (Prentice Hall), or *Standard C* by P. J. Plauger and Jim Brodie (Microsoft Press). If you're porting code, either from another compiler or from a previous version of THINK C, learn about writing portable code from a book like *Portability and the C Language* (Hayden Books) by Rex Jaeschke.

Contents

What Are the Standard Libraries? . . . 11
The ANSI libraries . . . 11
Handling Errors . . . 13
File-handling errors . . . 14
Using Files and Streams . . . 15
Using ANSI file functions . . . 16
Using UNIX file functions . . . 17
Naming files . . . 18
Binary and text files . . . 20

Allocating Memory . . . 20
How the ANSI library allocates memory . . . 21
Telling Time . . . 21
Numerical Limits and Parameters . . . 22
Floating-point parameters and limits . . . 22
Integer limits . . . 24
The Standard Library Functions . . . 26
Error functions . . . 26
I/O functions . . . 26
File-handling functions . . . 26
Formatted input and output functions . . . 27
Character input and output functions . . . 27
Direct input and output functions . . . 27
File positioning functions . . . 28
Error handling functions . . . 28
Console functions . . . 28
Math functions . . . 28
Floating point functions . . . 29
Integer functions . . . 29
Character functions . . . 30
Character testing functions . . . 30
Character converting functions . . . 30
String functions . . . 30
Copying functions . . . 30
Concatenating functions . . . 31
Comparing functions . . . 31
Searching functions . . . 31
Miscellaneous functions . . . 31
Time functions . . . 32
Telling time . . . 32
Converting time . . . 32
Writing time . . . 32
Miscellaneous functions . . . 32
Memory allocation functions . . . 32
Exiting functions . . . 33
Searching and sorting functions . . . 33
String conversion functions . . . 33
Variable argument list functions . . . 33
Jumping functions . . . 34
Signal-handling functions . . . 34
UNIX functions . . . 34
File functions . . . 34
Other functions . . . 34
Multibyte and wide character functions . . . 35
Locale functions . . . 35
OS functions . . . 36

What Are the Standard Libraries?

The `profile` library is also in the `C Libraries` folder. It's described in the THINK C User Manual, *Chapter 18, "The Profiler."*

THINK C includes two standard libraries, `ANSI` and `unix`, which you can find in the `C Libraries` folder:

- **The `ANSI` library** is a complete implementation of the ANSI standard library. The standard gives all C programmers in all ANSI C environments a consistent set of functions, making programs easier to port. It includes functions to perform file and screen I/O, string-handling, math, and more.
- **The `unix` library** contains several functions common to UNIX implementations of C that are not part of the ANSI standard library. It includes functions to handle files, signals, and other UNIX-specific features.

Use the `unix` library to make porting software from UNIX systems easier. When writing a new program, try to avoid the `unix` library and use in the `ANSI` library, instead. The descriptions of the `unix` functions in Chapter 5, "The Standard Library Functions," tell you when there is an equivalent function in the `ANSI` library. The most significant difference between the two libraries is how they handle files, which is described in "Using Files and Streams" on page 15.

The ANSI library includes the console package, which lets you use simple Macintosh windows called consoles. Consoles let you port MS-DOS and UNIX programs easily and and helps you get simple C programs running quickly. For more information, see Chapter 3, "Using Console Windows."

The ANSI libraries

These are special versions of the `ANSI` library for certain cases. They leave out some functions. Use these libraries only if your program meets the condition:

Use this library...	if your program...
`ANSI-small`	does not use consoles or floating point functions.
`ANSI-A4`	is a desk accessory, device driver, or other kind of code resource.

This is how the special ANSI libraries are different from the `ANSI` library:

- **The `ANSI-small` library** is smaller than `ANSI`, and projects that use it are smaller as well. It leaves out many things that are available from the Macintosh Toolbox. Most importantly, it leaves

out console functions and floating-point support. Also, the `%f`, `%g`, and `%e` conversions are not available for the formatted I/O functions, like `printf()` and `scanf()`.

- **The `ANSI-A4` library** is a subset of `ANSI-small` It leaves out everything that you can't use outside of applications. In addition to all the things `ANSI-small` leaves out, it leaves out file I/O functions. However, it includes string handling functions, like `sscanf()` and `sprintf()`. This library is for device drivers, desk accessories, and code resources only. The other standard libraries should be used in applications only.

The `ANSI`, `ANSI-small`, `ANSI-A4`, and `unix` libraries work with applications that turn on the "Generate MC68881 Code" option. Create a library specifically for machines with the co-processor if your application will run only on machines with the co-processor and you want it to run faster .

You may want to create your own version of the ANSI library. For example, you may want one for Macintoshes with an MC68881 math co-processor or one that includes floating-point functions and leaves out console functions. Here's how to do it:

1. Choose the library you want to start from. For example, if you want your library to exclude console and floating-point functions, start with `ANSI-small`.
2. In the Finder, create a copy of the the project and give it a new name, like `ANSI-881` or `MyANSI`.
3. In THINK C, open your project.
4. Choose **Options...** from the **Edit** menu and set up the options. Some common settings are listed below.
5. Remove files you don't need. For example, if you don't need time functions, remove `time.c`.
6. Choose **Bring Up To Date** from the **Project** menu to compile your project.

If you remove files from the `ANSI` library, you may get link errors when you compile your project. In the Link Errors window THINK C lists the functions that other functions in your project need. Search the files you removed , and add the files that contain the needed functions to your project.

For more information on setting the options, see the THINK C User Manual *Chapter 10, "The Compiler."*

Here are some common options settings:

To create a library that uses...	Set these options...
No console functions	In the Prefix section, #define the symbol `_NOCONSOLE_`, and remove the files `console.c` and `command.c`.
No floating-point functions	In the Prefix section, #define the symbol `_NOFLOATING_`.
The MC68881, directly	Turn on the "Generate 68881 instructions" option.
SANE floating-point format	Turn off the "Generate 68881 instructions" option, and turn on the "Native floating-point format" option.
MC68881 floating-point format	Turn on the "Generate 68881 instructions" option, and turn on the "Native floating-point format" option.

You can combine some of these options. For example, you can make a library for Macintoshes with the MC68881 that leaves out the console functions.

Handling Errors

Some functions in the standard libraries set the variable `errno` to an error code when they encounter errors. The file `errno.h` declares `errno` and defines macros for its possible values. If your program uses `errno` or those macros, it must #include `errno.h`.

Note

Before you use `errno` with a function, check the function's documentation to see if it sets `errno`. If the documentation doesn't say it sets `errno`, it may set `errno` even if there's been no error.

To see if a function causes an error, follow these steps:

1. Set `errno` to zero.
2. Call the function.
3. Check the value of `errno`. If it's not zero, an error happened.

This example shows you how to use `errno`:

```
#include <errno.h>

main()
{
    errno = 0;
    . . . /* Call the function. */
    if ( errno != 0 )
        . . . /* Handle your error. */
    else
        . . . /* There is no error. */
}
```

If you need to know what sort of error the function caused, check the value of `errno` against the macros in `errno.h`. Here are the two most common values for `errno`:

Value	Description
EDOM	A domain error. You passed an invalid value to a function.
ERANGE	A range error. The function generated an erroneous value.

Here are some other values that occur less frequently:

Value	Description
EINTR	Function aborted due to an interrupt
EINVAL	Invalid argument.
EMFILE	The ANSI library can't open more streams.
ENODEV	The device can't handle this operation (e.g., you're trying to set the file position for a console.)

The `unix` library also uses these values:

Value	Description
ESRCH	No such process
EBADF	Invalid file number

File-handling errors

The file-handling functions in the standard libraries call Macintosh Toolbox functions to open, close, read from, write to, and otherwise access files. If a Toolbox function returns an error, the standard library function that called it sets `errno` to that error code. For more information on what these codes

are, see section "File Manager Errors" in *Inside Macintosh III*, Appendix A, "Result Codes."

To help you port programs from MS-DOS and UNIX, THINK C assigns additional names to some Macintosh Toolbox error codes. These names are taken from error codes commonly found on MS-DOS and UNIX systems.

THINK C's extra name	Macintosh name	Description
EACCES	permErr	You don't have the proper access privileges.
EBUSY	fBsyErr	File is busy, can't delete it.
EEXIST	dupFNErr	Duplicate filename, can't rename a file to it.
EIO	ioErr	I/O error.
ENFILE	tmfoErr	The File Manager can't open more files.
ENOENT	fnfErr	File not found.
ENOSPC	dskFulErr	Disk full.
ENOTDIR	dirNFErr	Directory not found.
EROFS	wPrErr	Disk is write-protected

Using Files and Streams

This section gives you an overview of how the THINK C standard libraries handles files. It assumes you know how the ANSI standard handles files and emphasizes the differences between the THINK C implementation and implementations on UNIX and MS-DOS machines. For more detailed information, see a good C tutorial, such as *The C Programming Language, Second Edition* by Kernighan and Ritchie.

THINK C lets you handle I/O three ways. You can use the Macintosh File Manager , the ANSI functions, or the UNIX functions.

- **The Macintosh File Manager functions** let you take full advantage of the Macintosh interface and operating system. For more information, see *Inside Macintosh II*, Chapter 4, "The File Manager."
- **The ANSI functions** let you write portable programs. For more information, see "Using ANSI file functions" below.
- **The UNIX functions** let you port programs from UNIX systems that don't have ANSI libraries. For more information, see both "Using ANSI file functions" and "Using UNIX file functions" below.

Using ANSI file functions

The I/O functions in the standard libraries operate on streams. A **stream** is a place that your program can read characters from or write characters to. In THINK C, a stream can be either a file or a console window.

For more information on consoles, see Chapter 3, "Using Console Windows."

A console window is a Macintosh window that behaves like a simple terminal, the display that many MS-DOS and UNIX computers use. You type at a console window just like you type at a computer terminal.

There are other ways to open console windows. See `fopenc()` *on page 174.*

To use a console window as a stream, call an I/O function that uses one of the standard streams (`stdin`, `stdout`, or `stderr`). The `ANSI` library automatically creates a console window and uses it as the standard stream. For example, this program creates a console window and greets you:

```
#include <stdio.h>

main()
{
   printf( "Hello there.\n" );
}
```

To use a file as a stream, open it with `fopen()`. For example, this code fragment opens the file `MyFile` for reading:

```
FILE *fp;
fp = fopen( "MyFile", "r" );
```

Note

The ANSI I/O functions operate only the data fork of a Macintosh file. To operate on the resource fork, use the Macintosh Resource Manager.

The stream is a pointer to a structure, of type `FILE`, that contains information the library needs to know about your file or console.To refer to a file or a console in a function call, use its stream. For example, this statement reads a character from the file `fp`:

```
c = fgetc( fp );
```

And this example reads a character from your program's console:

```
c = fgetc( stdin );
```


Note

The ANSI functions, UNIX functions, and Macintosh Toolbox have different data structures and functions for handling files. You can't use Toolbox refnums in ANSI functions, and you can't use ANSI streams in Toolbox functions.

The standard libraries keep a table for all the open streams. It has slots for `FOPEN_MAX` streams, which `stdio.h` defines as `15`. When you open a new console or file, it takes up a slot in that table, so you can't have more than `FOPEN_MAX` files and consoles open at a time. However, files that you open with Macintosh File Manager don't take up slots in that table. If you try to open more than `FOPEN_MAX` streams with a standard library function, the function sets `errno` to `EMFILE`. If you try to open more files than the Macintosh File Manager can handle with a standard library function, the function sets `errno` to `ENFILE`.

Using UNIX file functions

The UNIX and ANSI I/O functions refer to streams differently. The UNIX functions refer to a stream with a file number instead of a pointer to a structure of type `FILE`. A **file number** is a small positive integer assigned to a stream. It refers to the slot that the stream occupies in the standard libraries' table of streams. It is not related to a Macintosh refnum. For compatibility with UNIX and MS-DOS, the file numbers `0`, `1`, and `2` are reserved, as follows:

This file number...	**refers to this stream...**
0	`stdin`
1	`stdout`
2	`stderr`

Note

Many UNIX implementations use *file descriptor* to refer to the number assigned to each open file. This manual calls that number the *file number*.

Use the UNIX I/O functions to make porting programs from UNIX easier. Don't use them in new programs. Instead, use the ANSI or Macintosh Tool-

box functions. These UNIX I/O functions are roughly equivalent to these ANSI functions:

Unix library	ANSI library
open()	fopen()
close()	fclose()
read()	fread()
write()	fwrite()
fdopen()	fileno()
creat()	fopen()
lseek()	fseek()
tell()	ftell()

For more information on a function, see its description in Chapter 5, "The Standard Library Functions."

Naming files

On the Macintosh, file names can have up to 31 characters. You can use any printing characters from the Macintosh character set except for the colon (:). That means you can use upper and lower case characters, international characters (like ä or ß), and characters that aren't alphanumeric (like # or •). However, the Macintosh ignores case when comparing file names. For example, the Macintosh considers `MyFile` and `myfile` to be the same file.

When your program refers to a file by its name, the ANSI library looks for it in the default folder. When you start an application, the default folder is the folder your application is in. For example, look at what's on the disk in Figure 2-1.

When you run `MyProg`, the default folder is `MyProg folder`. If you look for the file `Gadgets`, the ANSI library can find it. But if you look for `Widgets`, the ANSI library can't.

To find a file that's in a different folder from the default, you have to use a **pathname** to specify where your file is. A pathname can have up to 255 characters. A **full pathname** contains the name of the disk your file is on and the folders you have to open to reach your file. The disk name and folder names are separated with colons (:). For example, the full pathname for `Bob` is

```
MyDisk:Letters:Bob
```

And the full pathname for `Widgets` is

```
MyDisk:MyProg folder:My Docs:Widgets
```


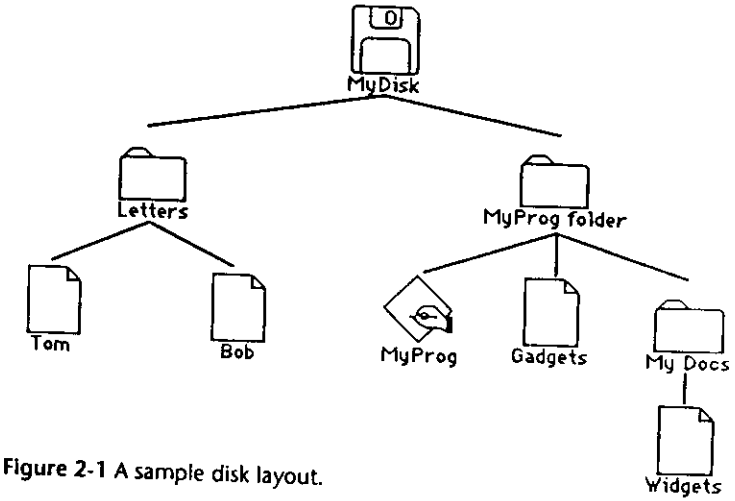

Figure 2-1 A sample disk layout.

Full pathnames get cumbersome especially on a hard drive. And they quickly run over the maximum length. Fortunately, there's an alternative.

A **partial pathname** specifies how to reach a file from the default folder. It must start with a colon (:). (Remember that when you start an application, the default folder is the folder the application is in.) For example, the partial pathname for `Widgets` is

```
:My Docs:Widgets
```

Partial pathnames can also refer to files that aren't in folders in the default folder. For example, `Bob` is in `Letters`, which isn't in `MyProg folder`. To move to the folder (or disk) that contains the default folder, use two colons (::). So, the partial pathname for `Bob` is

```
::Letters:Bob
```

You can move to folders further up the tree by using more colons, one for each folder you're moving up. For example, if `My Docs` were the default folder, the partial pathname for `Bob` would be

```
:::Letters:Bob
```

You can change the default folder with the Toolbox function `PBHSetVol()`. *For more information, see* Inside Macintosh IV, *Chapter 19, "File Manager."*

Binary and text files

To see how to open a file as a text or binary file, see `fopen()` *on page 171.*

The THINK C standard libraries let you open a file as either a text file or a binary file. They make the contents of a text file look like a series of characters separated into lines. They make the contents of a binary file look like binary data, such as a file of records or numerical information.

On most computer systems, like UNIX and MS DOS, newlines (`'\n'` or `0x0A`) separate the lines in text files. But on the Macintosh, carriage returns (`'\r'` or `0x0D`) separate the lines. To make porting programs from UNIX and MS-DOS easier, the standard libraries make it look like newlines separate lines. When you read from a text file, the standard libraries convert carriage returns to newlines. When you write to a text file, the standard library converts newlines to returns.

Note

The standard libraries perform no conversion on binary files.

For example, this statement writes the string `"Hello world\r"` to the text file `fp`:

```
fputs( "Hello world\n", fp );
```

And if the text file `fp` contains `"Hi there\r"`, this statement puts the string `"Hi there\n"` into `s`:

```
fgets( s, 80, fp );
```

Allocating Memory

THINK C lets you allocate memory with the ANSI standard functions `malloc()`, `calloc()`, `realloc()`, and `free()`. Use these functions if you're porting a program or writing portable code. But if your program runs only on the Macintosh, consider using the Toolbox Memory Manager instead. Most importantly, don't use these functions together with the Memory Manager. Since these two systems handle memory differently, using them together leads to the most inefficient use of memory.

When used alone, the Macintosh Memory Manager is more efficient since it can combine blocks of freed memory together to make larger blocks. For more information on the Memory Manager, see *Inside Macintosh II*, Chapter 1, "Memory Manager."

How the ANSI library allocates memory

This section is a technical description of how THINK C's `ANSI` library functions `malloc()`, `calloc()`, and `realloc()` allocate memory. It assumes you have some knowledge of the Memory Manager. Be warned that these functions may allocate memory differently in later versions of THINK C and do not write code that relies on this implementation.

The ANSI library allocates blocks of non-relocatable memory with the Toolbox function `NewPtr()`. It always allocates blocks that are at least a minimum size, 15,000 bytes. (That size is also the maximum size for a request that can be fulfilled without calling `NewPtr()` and is therefore called `MAXSIZ`.)

When you call one of the allocation functions, it checks how much you want. If you want a block larger than 15,000 bytes, it uses `NewPtr()` to allocate a block exactly the size you want and returns it. If you want a block smaller than 15,000 bytes, it uses `NewPtr()` to allocate a 15,000-byte block, saves it, and returns a pointer to part of it. The next time you want a block smaller than 15,000 bytes, it checks if there's enough space in the block it saved. If there is, it returns a pointer to another part of the block. Otherwise, it allocates a new block.

Telling Time

The THINK C standard libraries let you handle time three ways: clock time, calendar time, and broken-down time.

Clock time is useful for measuring the duration of tasks that happen while your program is running. In THINK C, clock time is the number of clock ticks since the system was last started. (There are 60 clock ticks per second.) It's of type `clock_t`. To get the current clock time, call `clock()`.

Calendar time is useful for telling the time of day or the day of the year. In THINK C, calendar time is the number of seconds since Midnight, January 1, 1904. It's of type `time_t`. To get the current calendar time, call `time()`.

Broken-down time is useful for handling dates and times. Broken-down time a structure of type `struct tm` that includes these fields: (They're all of type `int`.)

Field	Description (range)
`tm_sec`	Seconds after the minute (0 to 61)
`tm_min`	Minutes after the hour (0 to 59)
`tm_hour`	Hours since midnight (0 to 23)
`tm_mday`	Day of the month (1 to 31)
`tm_mon`	Months since January (0 to 11)
`tm_year`	Years since 1900
`tm_wday`	Days since Sunday (0 to 6)
`tm_yday`	Days since January 1 (0 to 365)
`tm_isdst`	Daylight Savings Time flag

To get the current broken-down time, call `mktime(time(NULL))`.

Numerical Limits and Parameters

To see how THINK C prints special floating-point values, like Infinity or NaN (Not a Number), see "Printing Infinity and NaNs" on page 68.

This section describes some limits and parameters for THINK C's numbers, such as the largest possible value and the smallest possible fraction. These parameters are defined as macros in the header files `float.h` and `limits.h`.

Changing the values of the macros won't change the way THINK C represents numbers. The values are listed here to let you know the limits and attributes of THINK C's numbers.

Warning

Don't use the value of a macro in your source code. The value might change in later versions of THINK C. Use the macro instead.

Floating-point parameters and limits

These parameters are defined in `float.h` as macros. Other implementations of C could have different values.

For more information on how the Macintosh represents floating-point numbers, see the Apple Numerics Manual, Second Edition *(Addison-Wesley).*

These macros define attributes for the THINK C implementation of floatin point numbers.

Macro	Description
`FLT_RADIX`	The radix for exponent representation. THINK C uses 2.
`FLT_ROUNDS`	Specifies how floating-point functions round the results of addition. THINK C rounds to the nearest floating-point value.

The macros in the next chart define limits and parameters for the THINK C implementation of floating-point numbers. The first few letters of the macr tell you the type of numbers they're for, as shown here:

Macros starting with...	Describe limits for...
`FLT`	`float`
`DBL`	`double`
`LDBL`	`long double`

For more information on the Options... dialog, see the THINK C User Manual, *Chapter 10, "The Compiler."*

The "8-byte double" option in the Compiler Settings page of the **Options...** dialog changes the values of the macros beginning with `DBL`. This chart lists their values if that option is off (the default setting). The chart after this one lists their values if that option is. In both charts, all floating-point values are rounded to five decimal places.

Note

The settings of the "Generate 68881 instructions" and "Native floating-point format" options in the Compiler Settings page of the **Options...** dialog do *not* change the values of these macros.

Macro	Value	Description
`FLT_MANT_DIG`	24	The number of base
`DBL_MANT_DIG`	64	`FLT_RADIX` digits in the
`LDBL_MANT_DIG`	64	floating-point significand
`FLT_DIG`	6	Decimal digits of precision
`DBL_DIG`	18	
`LDBL_DIG`	18	

Macro	Value	Description
FLT_MIN_EXP	-125	The largest integer that can be a negative exponent when the radix is FLT_RADIX.
DBL_MIN_EXP	-16,382	
LDBL_MIN_EXP	-16,382	
FLT_MIN_10_EXP	-37	The largest integer that can be a negative exponent, when the radix is 10.
DBL_MIN_10_EXP	-4931	
LDBL_MIN_10_EXP	-4931	
FLT_MAX_EXP	128	The largest integer that can be a positive exponent, when the radix is FLT_RADIX.
DBL_MAX_EXP	16,384	
LDBL_MAX_EXP	16,384	
FLT_MAX_10_EXP	38	The largest integer that can be a positive exponent, when the radix is 10.
DBL_MAX_10_EXP	4932	
LDBL_MAX_10_EXP	4932	
FLT_MAX	3.40282E+38	The largest representable floating-point number.
DBL_MAX	1.18973E+4932	
LDBL_MAX	1.18973E+4932	
FLT_EPSILON	1.19209E-7	The smallest fraction that can be represented in floating point notation.
DBL_EPSILON	1.08420E-19	
LDBL_EPSILON	1.08420E-19	
FLT_MIN	1.17549E-38	The smallest normalized positive floating-point number.
DBL_MIN	1E-4926	
LDBL_MIN	1E-4926	

If you turn on the "8-byte doubles" option, these are the values for the macros beginning with DBL.

Macro	Value
DBL_MANT_DIG	53
DBL_DIG	15
DBL_MIN_EXP	-1021
DBL_MIN_10_EXP	-307
DBL_MAX_EXP	1024
DBL_MAX_10_EXP	308
DBL_MAX	1.79769E+308
DBL_EPSILON	2.22045E-16
DBL_MIN	2.22507E-308

Integer limits

This section describes limits for the THINK C implementation of integers: the largest and smallest values for each integer type. These parameters are defined in limits.h as macros. Other implementations of C could have different values.

For more information on the Options... dialog, see the THINK C User Manual, *Chapter 10, "The Compiler."*

The following chart does not list the values for INT_MIN, INT_MAX, and UINT_MAX. The values of these macros depends on the setting of the "4-byte int" option in the Compiler Settings page of the **Options...** dialog. If that option is off (the default setting), those macros have these values:

If "4-byte ints" is off, this macro...	has the same value as this macro...
INT_MIN	SHRT_MIN
INT_MAX	SHRT_MAX
UINT_MAX	USHRT_MAX

If that option is on, those macros have these values:

If "4-byte ints" is on, this macro...	has the same value as this macro...
INT_MIN	LONG_MIN
INT_MAX	LONG_MAX
UINT_MAX	ULONG_MAX

This chart lists the values for the rest of the macros:

Macro	Value	Description
CHAR_BIT	8	Number of bits for the smallest object that is not a bit-field.
MB_LEN_MAX	1	The maximum number of characters that can be in a multi-byte character.
SCHAR_MIN	-128	Minimum value for a signed char
SCHAR_MAX	127	Maximum value for a signed char
UCHAR_MAX	255	Maximum value for an unsigned char
CHAR_MIN	-128	Minimum value for a char

Macro	Value	Description
CHAR_MAX	127	Maximum value for a `char`
SHRT_MIN	-32,768	Minimum value for a `short int`
SHRT_MAX	32,767	Maximum value for a `short int`
USHRT_MAX	65,535	Maximum value for an `unsigned short int`
LONG_MIN	-2,147,483,648	Minimum value for a `long int`
LONG_MAX	2,147,483,647	Maximum value for a `long int`
ULONG_MAX	4,294,967,295	Maximum value for an `unsigned long int`

The Standard Library Functions

This section lists all the functions in the ANSI library, by category. To find more information on any function, see Chapter 5, "The Standard Library Functions."

Some of these functions are also implemented as macros to give you the advantages of both. When you use one as an argument to a function, THINK C passes a pointer to the function. When you put one in a statement, THINK C uses the macro, which will run faster.

Error functions

These functions write error messages and diagnostic messages. (`assert()` is implemented as a macro.)

```
void  assert(int expression)
void  perror(const char *string);
char *strerror(int errnum);
```

I/O functions

These functions let you read from and write to files and console windows.

File-handling functions

These functions let you handle files: opening, closing, renaming, deleting, and more.

```
int   fclose(FILE *stream);
int   fflush(FILE *stream);
FILE *fopen(const char *name, const char *mode);
FILE *freopen(const char *name, const char *mode,
              FILE *stream);
int   remove(const char *name);
int   rename(const char *old, const char *new);

void  setbuf(FILE *stream, char *buf);
int   setvbuf(FILE *stream, char *buf, int mode,
              size_t size);
FILE *tmpfile(void);
char *tmpnam(char *name);
```

Formatted input and output functions

These functions read and write formatted data to streams. For more information, see Chapter 4, "Using printf and scanf."

```
int fprintf(FILE *stream, const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
int printf(const char *format, ...);
int scanf(const char *format, ...);
int sprintf(char *string, const char *format, ...);
int sscanf(const char *string,
           const char *format, ...);
int vfprintf(FILE *stream, const char *format,
             va_list args);
int vprintf(const char *format, va_list args);
int vsprintf(char *string, const char *format,
             va_list args);
```

Character input and output functions

These functions read and write characters to streams. (`getc()`, `getchar()`, `putc()`, and `putchar()` are implemented as both functions and macros.)

```
int   fgetc(FILE *stream);
char *fgets(char *string, int nchars, FILE *stream);
int   fputc(int c, FILE *stream);
int   fputs(const char *string, FILE *stream);
```

```
int   getc(FILE *stream);
int   getchar(void);
char *gets(char *string);
int   putc(int c, FILE *stream);
int   putchar(int c);
int   puts(const char *string);
int   ungetc(int c, FILE *stream);
```

Direct input and output functions

These functions read and write binary data to streams.

```
size_t fread(void *buf, size_t size,
             size_t nmemb, FILE *stream);
size_t fwrite(const void *buf, size_t size,
              size_t nmemb, FILE *stream);
```

File positioning functions

These functions set or record the current position for a file.

```
int  fgetpos(FILE *stream, fpos_t *pos);
int  fseek(FILE *stream, long offset, int whence);
int  fsetpos(FILE *stream, const fpos_t *pos);
long ftell(FILE *stream);
void rewind(FILE *stream);
```

Error handling functions

These functions let your clear errors or tell you about errors that happened when using files. (`feof()` and `ferror()` are implemented as both functions and macros.)

```
void clearerr(FILE *stream);
int  feof(FILE *stream);
int  ferror(FILE *stream);
void perror(const char *string);
```

Console functions

These functions let you manipulate consoles, windows that act like simple display terminals. For more information on using consoles, see Chapter 3, "Using Console Windows." All these functions are in the ANSI library, except for `isatty()`, which is in the `unix` library.

```
void  ccleol(FILE *fp);
void  ccleos(FILE *fp);
int   ccommand(char ***p);
```



```
void  cecho2file(char *file, int append, FILE *fp);
void  cecho2printer(FILE *fp);
void  cgetxy(int *x, int *y, FILE *fp);
void  cgotoxy(int x, int y, FILE *fp);
void  chide(FILE *fp);
void  cinverse(int x, FILE *fp);
void  csetmode(int mode, FILE *fp);
void  csettabs(int n, FILE *fp);
void  cshow(FILE *fp);
FILE *fopenc(void);
FILE *freopenc(FILE *fp1, FILE *fp2);
int   isatty(int fn);
```

Math functions

These are the math functions. All of them are in the ANSI library.

Floating point functions

These are math functions for floating-point numbers.

```
double acos(double x);
double asin(double x);
double atan(double x);
double atan2(double x, double y);
double ceil(double x);
double cos(double x);
double cosh(double x);
double exp(double x);
double fabs(double x);
double floor(double x);
double fmod(double x, double y);
double frexp(double x, int *exp);
double ldexp(double x, int exp);
double log(double x);
double log10(double x);
double modf(double x, double *ip);
double pow(double x, double y);
double sin(double x);
double sinh(double x);
double sqrt(double x);
double tan(double x);
double tanh(double x);
```

Integer functions

These are the math functions for integers.

```
int     abs(int i);
div_t   div(int numer, int denom);
long    labs(long j);
ldiv_t  ldiv(long numer, long denom);
int     rand(void);
void    srand(unsigned int seed);
```

Character functions

These functions test and convert characters. All of them are in the ANSI library.

Character testing functions

These functions test whether a character is of a certain kind. (These are implemented as both functions and macros.)

```
int isalnum(int c);
int isalpha(int c);
int iscntrl(int c);
int isdigit(int c);
int isgraph(int c);
int islower(int c);
int isprint(int c);
int ispunct(int c);
int isspace(int c);
int isupper(int c);
int isxdigit(int c);
```

Character converting functions

These functions convert a character from upper to lower case, or vice versa.

```
int tolower(int c);
int toupper(int c);
```

String functions

These functions perform operations on blocks of memory and strings. In general, the functions starting with `str` work only with null-terminated strings, and the functions starting with mem work with any block of memory.

Copying functions

These functions copy data from one object to another.

```
void *memcpy(void *to, const void *from, size_t size);
void *memmove(void *to, const void *from,
              size_t size);
char *strcpy(char *to, const char *from);
char *strncpy(char *to, const char *from,
              size_t size);
```

Concatenating functions

These functions append one object to another.

```
char *strcat(char *to, const char *from);
char *strncat(char *to, const char *from,
              size_t size);
```

Comparing functions

These functions compare one object to another.

```
int memcmp(const void *s1, const void *s2,
           size_t size);
int strcmp(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2,
            size_t size);
```

Searching functions

These functions search for characters in an object.

```
void   *memchr(const void *obj, int c, size_t size);
char   *strchr(const char *string, int c);
size_t  strcspn(const char *string, const char *set);
char   *strpbrk(const char *string, const char *set);
char   *strrchr(const char *string, int c);
size_t  strspn(const char *string, const char *set);
char   *strstr(const char *string,
               const char *substring);
char   *strtok(char *string, const char *delims);
```

Miscellaneous functions

These functions let you initialize objects, create a string with an error message, and figure the length of a string.

```
void   *memset(void *obj, int c, size_t size);
char   *strerror(int errnum);
size_t  strlen(const char *string);
```

Time functions

These functions handle time. All of them are in the ANSI library.

Telling time

These functions tell you the current time.

```
clock_t clock(void);
time_t  time(time_t *caltime);
```

Converting time

These functions convert time.

```
struct tm *localtime(const time_t *caltime);
time_t     mktime(struct tm *bdtime);
```

Writing time

These functions write the time to a string, in a format people can read.

```
char   *asctime(const struct tm *bdtime);
char   *ctime(const time_t *caltime);
size_t  strftime(char *string, size_t size,
                 const char *format,
                 const struct tm *bdtime);
```

Miscellaneous functions

These functions let you manipulate time in other ways

```
double     difftime(time_t caltime1, time_t caltime2);
struct tm *gmtime(const time_t *caltime);
```


Memory allocation functions

These functions let you allocate and deallocate blocks of memory. All of them are in the ANSI library.

```
void *calloc(size_t nmemb, size_t size);
void  free(void *ptr);
void *malloc(size_t size);
void *realloc(void *ptr, size_t size);
```

Exiting functions

These functions let you exit your program and control what happens when your program exits. All of them are in the ANSI library.

```
void abort(void);
int  _atexit(void (*func)());
int  atexit(void (*func)());
void _exit(int status);
void exit(int status);
void _exiting(int status);
```

Searching and sorting functions

These functions let you search and sort an array. All of them are in the ANSI library.

```
void *bsearch(const void *key, const void *array,
              size_t nmemb, size_t size,
              int (*compar)(const void *key,
                            const void *dat));

void  qsort(void *array, size_t nmemb, size_t size,
            int (*compar)(const void *key,
                          const void *dat));
```

String conversion functions

These functions let you convert the number in a string to an integer or floating-point value.

```
double         atof(void char *string);
int            atoi(const char *string);
long           atol(const char *string);
double         strtod(const char *string, char **end);
long           strtol(const char *string, char **end,
                      int base);
unsigned long strtoul(const char *string,
                      char **end, int base);
```

Variable argument list functions

These macros let you create functions that can take any number of arguments. (These are implemented are macros.) All of them are in the ANSI library.

```
void va_start(va_list ap, parmN)
type  va_arg(va_list ap, type);
void va_end(va_list ap);
```

Jumping functions

These macros let you make non-local jumps; for example, from one function to another function further up in the call chain. All of them are in the ANSI library.

```
int  setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);
```

Signal-handling functions

These functions let you raise and respond to signals. All of them are in the ANSI library.

```
void (*signal (int sig,void (*func)(int)))(int);
int  raise(int sig);
```

UNIX functions

These functions are commonly found in UNIX implementations of C but were not included in the ANSI standard library. All of them are all in the unix library.

File functions

These functions let you port code that use UNIX file functions. Whenever possible in new programs, use the ANSI functions described in section "Using UNIX file functions" on page 17.

```
int   close(int fn);
int   creat(char *filename, int mode);
FILE *fdopen(int fn, char *mode);
int   fileno(FILE *fp);
int   getw(FILE *stream);
int   isatty(int fn);
long  lseek(int fn, long offset, int whence);
int   open(char *filename, int mode);
int   putw(int word, FILE *stream);
int   read(int fn, char *buffer, unsigned int count);
```



```
long  tell(int fn);
int   unlink(char *filename);
int   write(int fn, char *buffer, unsigned nbytes);
```

Other functions

These functions simulate a UNIX environment for your program by returning values similar to those that might be returned on a UNIX system.

```
int   exec(char *path, ...);
int   execl(char *path);
int   execle(char *path);
int   execv(char *path, char argv[]);
int   execve(char *path, char *argv[], char *envp[]);
int   execvp(char *path, ...);
int   execvpe(char *path, ...);

int   getegid(void);
int   geteuid(void);
int   getgid(void);
int   getpgrp(void);
int   getpid(void);
int   getppid(void);
int   getuid(void);
char *getlogin(void);
int   kill(int pid, int sig);
void  sleep(int secs);
```

To use characters that are more than one-byte long, use the Script Manager, described in Inside Macintosh VI, *Chapter 14, "Worldwide Software Overview."*

Multibyte and wide character functions

These functions are for multibyte and wide characters. However, THINK C provides only minimal support for multibyte or wide characters. The maximum length for a multibyte character is one byte. All of them are in the ANSI library.

```
int    mblen(void char *mbc, size_t n);
size_t mbstowcs(wchar_t *wcs, const char *mb,
               size_t n);
int    mbtowc(wchar_t *wc, const char *mbc, size_t n);
size_t wcstombs(char *mbcs, const wchar_t *wcs,
               size_t n);
int    wctomb(char *mbc, wchar_t wc);
```

To localize your software for other countries, use the International Utilities Package, described in Inside Macintosh VI, *Chapter 14, "Worldwide Software Overview."*

Locale functions

These functions are for handling locales. However, THINK C provides only minimal support for locales. It supports only the minimal "C" environment. All of them are in the ANSI library.

```
struct lconv *localeconv(void);
char         *setlocale(int cat,const char *locale);
int           strcoll(const char *s1, const char *s2);
size_t        strxfrm(const char *s1, const char *s2,
                      size_t size);
```

To ask questions of the Macintosh system, use the Gestalt Manager, described in Inside Macintosh VI, *Chapter 3, "Compatibility Guidelines."*

OS functions

These functions are for asking questions of the operating system. THINK C provides only minimal support for them. They both return NULL. They are in the ANSI library.

```
char *getenv(const char *name);
int   system(const char *command);
```

These two functions return the the current user's name and run an application. They're in the unix library.

```
char *getlogin(void);
int   exec(char *path);
```


Using Console Windows

3

Programs that don't have a graphical interface can be difficult to port to the Macintosh. THINK C's console package helps you create or port these programs by creating a window that acts like the display of an MS-DOS or UNIX computer. This chapter describes the package and includes a complete listing of all the functions it contains.

Contents

What Is the Console Package? 39

Using the Console Package 40
- Pausing a program when it's done 41
- Handling Command-. (Command-Period) 41
- Using your windows in the console interface 43
- Using consoles outside the console interface 44

Modifying Console Windows 44
- Setting console options 44
- Setting the input mode 45
- Using inverse video 46

What Is the Console Package?

The console package lets you port UNIX or MS-DOS programs to the Macintosh easily and helps you get simple C programs running quickly. It's an extension to the ANSI standard library that's part of the THINK C ANSI library.

TTY is an abbreviation for teletypewriter.

The console package lets you use simple windows, called **consoles**, to write to, read from, create, move around and more. A console is a Macintosh window that behaves like a simple display terminal, the kind of terminal that many MS-DOS and UNIX computers use. For example, the simple "hello world!" program in *THINK C User Manual*, Chapter 3, uses a console, as shown in Figure 3-1.

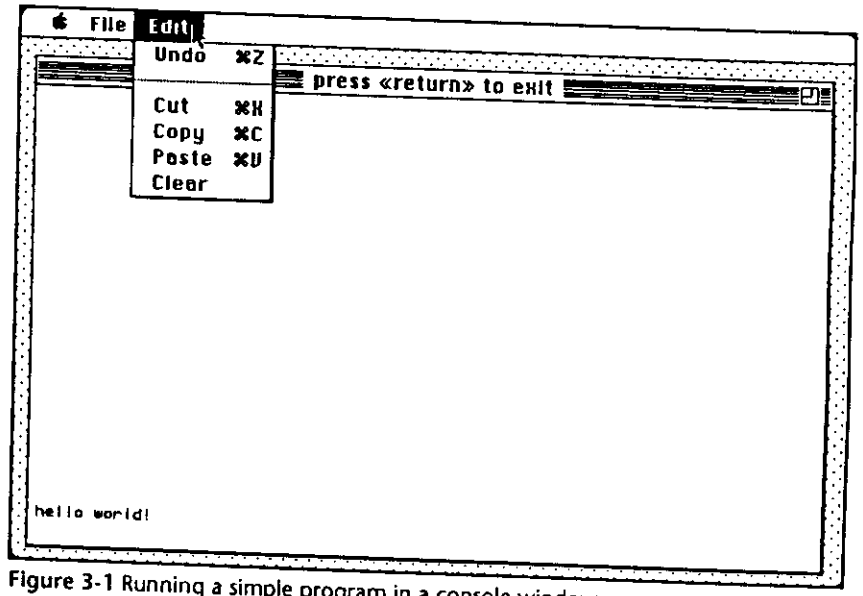

Figure 3-1 Running a simple program in a console window.

The console package also helps you port programs that require command-line arguments. The `ccommand()` function brings up a dialog, shown in Figure 3-2, that lets you enter a command line. You'll also find it useful for

redirecting input and output for other programs you write. For more information, see `ccommand()` on page 122.

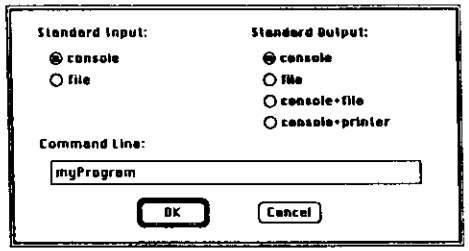

Figure 3-2 The ccommand() dialog.

Using the Console Package

If your program doesn't have a Macintosh interface, the console package creates one for you. Just include the `ANSI` library in your project. When you use one of the standard streams (`stdin`, `stdout`, or `stderr`), you trigger the console package to use the console interface. It does all this for you:

- Initializes the Macintosh Toolbox managers
- Creates the **Apple**, **File**, and **Edit** menus, shown in Figure 3-3
- Makes a console for you to write to and read from

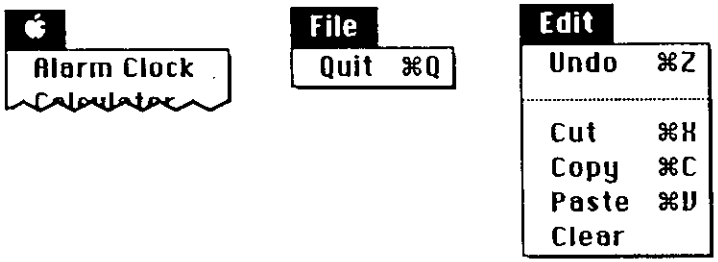

Figure 3-3 The menus for the generic console interface

Note

If you want to create your own interface and menu bar, you can still create consoles to write to. See "Using consoles outside the console interface" on page 44.

While your program is running, you can use desk accessories, cut and paste from a console window, and move it around. You can quit at any time by choosing **Quit** from the **File** menu or pressing Command-. (Command-Period).

To write to or read from the console, use the functions described in Chapter 5, "The Standard Library Functions." The console package uses the console for the standard streams (`stdin`, `stdout`, and `stderr`). To move the cursor inside the console, create new consoles, and perform other advanced tasks, use the functions described in this chapter.

Pausing a program when it's done

The console interface pauses your program when it's finished running. After the last statement in your program executes, the console environment brings the last console created to the front, changes its title to "press <return> to exit", and won't exit until you do one of the following:

`console_options` is a global variable that controls the appearance and behavior of console windows. For more information see section "Setting console options" on page 44

- Press Return or Enter
- Press Command-. (Command-Period).
- Choose **Quit** from the **File** menu.

If you want your program to exit immediately when it's finished running, change the value of `console_options.pause_atexit` to 0. By default, its value is 1.

Handling Command-. (Command-Period)

For more information on `signal()`, see page 265

When you press Command-. (Command-Period) in the console interface, your program quits. You can change this behavior, however, with the function `signal()`.

When you press Command-Period, the console interface raises the signal `SIGINT`. By default, the console interface responds to `SIGINT` by quitting your program. With the function `signal()`, you can tell the interface to respond differently: either to ignore the signal or to call a function. Your function might abort the current operation or ask if the user really wants to quit.

Note

The console interface responds to a Command-Period only when your program is writing to or reading from a console.

To ignore Command-Period, put this statement in your program:

```
signal( SIGINT, SIG_IGN );
```

To call a function, replace SIG_IGN in the statement above with your function. For example, this statement tells the console interface to call OKExit() when the user presses Command-Period.

```
signal( SIGINT, OKExit );
```

Before the console interface calls your function, it sets the response for SIGINT back to quitting your program. So, be sure your function calls signal(), or else the next time the user presses Command-Period, your program will quit.

This example prints the file MyFile to a console window. When the user presses Command-Period, it asks if the user really wants to quit:

```
#include <stdio.h>
#include <console.h>
#include <signal.h>

/* To be portable, define the flag that */
/* the signal handler sets like this.   */
volatile sig_atomic_t sigFlag;

/* The signal handler should be a short    */
/* function that resets the signal and     */
/* sets a global flag.                     */
void sigHandler( int sig )
{
   signal( SIGINT, sigHandler );
   sigFlag = 1;
}

/* This function makes sure   */
/* the user wants to exit.    */
int OKExit()
{
   char c;

   do {
      printf( "\a\n\n" );
      printf( "Do you want to exit? (y or n) ");
      c = getchar();
   } while ( (c!='y') && (c!='n') );
   printf( "\n\n" );
   if (c=='y')
      return 1;
   else {
      sigFlag = 0;
      return 0;
   }
}

main()
{
   char c;
   FILE *fp;

   /* Set up the console window. */
   cshow(stdout);
   csetmode(C_CBREAK, stdout);
   console_options.pause_atexit = 0;

   /* Set up the SIGINT response. */
   signal( SIGINT, sigHandler );
   sigFlag = 0;

   /* Open MyFile and print it. */
   fp = fopen( "MyFile", "r" );
   for ( c=getc(fp); c!=EOF; c=getc(fp) ) {
      putchar( c );

      /* If the user pressed Command-Period  */
      /* and wants to exit, stop printing    */
      /* the file.                           */
      if (sigFlag && OKExit())
         break;
   }
   fclose( fp );
}
```

Using your windows in the console interface

You can use your own windows with the console package interface, but you must use a standard stream to trigger the console package to initialize the Macintosh Toolbox managers and use the console interface. Be sure you use one of the standard streams *before* you use your own window. For example, this program would crash without calling cshow():

```
cshow( stdout );
DrawingWindow = NewWindow(NULL, . . . );
printf( "Create [O]val or [R]ectangle " );
```

Using consoles outside the console interface

If you don't use the console interface, you might want to use consoles as simple windows you can write to. To create a console window, call `fopenc()`, described on page 184. Remember that you can only write to these consoles — you cannot read from them. You must create your own windows if you want input from the user.

Be sure you initialize the Macintosh Toolbox before you call `fopenc()`. The console package decides whether to use its own interface by checking whether you've initialized the Toolbox.

Modifying Console Windows

This section explains how you can control how your console windows appear and behave.You can set some console options, change a console's input mode, and use inverse video.

Setting console options

Before you open a console window, you can set some of its attributes, like its size, position, and type face. Just change the values in the global structure `console_options`. The functions `fopenc()` and `freopenc()` set the attributes of a new console according to that structure.

Note

You can't change a console's attributes after you create it.

These are the fields in `console_options` defined in `console.h`: (The members are of type `short int`, unless otherwise specified.)

Field	Description (default)
`top`	Position of the top (50)
`left`	Position of the left side (10)
`title`	Window's title, a Pascal string of type `unsigned char * ("console")`
`txFont`	Font used to write text (4, Monaco)
`txSize`	Size of the text, in points (9)
`txFace`	Style of the text, bold, italic, etc. (0, Plain)
`nrows`	Number of lines of text (25)
`ncols`	Number of characters per line (80)
`procID`	The type of window (8)
`pause_atexit`	Whether to pause a program when it's finished running (1, True)

Note

The field `pause_atexit` doesn't affect the next console opened. It lets you change how your program behaves when it exits. See "Pausing a program when it's done" on page 41 for more information.

If you use the console interface, you can still set the attributes of the console it creates. Just modify `console_options` before you use one of the standard streams. For example, this code fragment shrinks the size of the console before it asks you a question:

```
console_options.nrows = 5;
printf( "Enter your name: " );
```

Setting the input mode

There are four ways the console package handles input. The default mode is `C_ECHO`.

Mode	Buffering	Echoing	Editing
`C_ECHO`	line-buffered	echoed	full
`C_NOECHO`	line-buffered	none	partial
`C_CBREAK`	none	echoed	none
`C_RAW`	none	none	none

To set the input mode, use the function `csetmode()`. For example, this statement sets the input mode for standard input to `C_RAW`:

```
csetmode( C_RAW, stdin );
```

The most important attribute is buffering. The line-buffered modes let you edit a line of text before entering it. The unbuffered modes do not.

The line-buffered modes are `C_ECHO` and `C_NOECHO`. These modes give a line to the program when you press Return or Enter. Before entering the line, you can use these keys to edit it:

To do this...	Use this key...
Delete a character	Delete
Erase a line	Escape, Clear, Command-U, Command-Z, Control-U, or Control-Z
Enter end-of-file	Command-D or Control-D
Enter a line	Return or Enter

C_ECHO and C_NOECHO differ in the following ways:

- C_ECHO mode echoes each character that you type as you type it, so you always know what you typed. C_NOECHO doesn't, so you don't know what you typed unless the program writes it out.
- In addition to the keys mentioned above, C_ECHO lets you edit with standard Macintosh commands, like **Cut**, **Copy**, and **Paste**, the arrow keys, and so on. C_NOECHO doesn't let you use these additional techniques.

The unbuffered modes are C_RAW and C_CBREAK. They do not buffer what you type, but give the program what you type immediately.

C_RAW and C_CBREAK differ in the following ways:

- C_RAW mode doesn't echo a character when you type it. C_CBREAK does.
- C_RAW mode lets your program see command keys. It converts them to control characters. For example, if you press Command-A, the program sees Control-A (or 0x01). C_CBREAK mode doesn't convert command keys.
- C_RAW mode lets you program see arrow keys. It passes 0x1E for the Up Arrow, 0x1F for the Down Arrow, 0x1C for the Left Arrow, and 0x1D for the Right Arrow. C_BREAK mode does not pass the arrow keys to your program.
- In C_RAW mode, input requests aren't blocked. When a program calls `getchar()`, it returns immediately if you haven't typed something. C_CBREAK mode waits for you to type something.
- C_CBREAK mode ignores Tabs. C_RAW mode does not.

Using Inverse Video

In a console window, you can write out characters in inverse video; that is, white characters on a black background. This feature lets you port programs that take advantage of a terminal with inverse video.

To write an inverted character, turn on the inverse video mode with `cinverse()` and write the character with its high bit set. To set its high bit,

OR it with 0x80. For example, this code writes what you see in Figure 3-4: *hello* in inverse video and *world* in normal video:

```
cinverse (TRUE, stdout);
putchar ('h'|0x80);
putchar ('e'|0x80);
putchar ('l'|0x80);
putchar ('l'|0x80);
putchar ('o'|0x80);
putchar (' ');
putchar ('w');
putchar ('o');
putchar ('r');
putchar ('l');
putchar ('d');
```


Figure 3-4 *hello* in inverse video, *world* in normal video.

When you turn on the inverse video mode, keep these things in mind:

- You cannot write `'Ä'` (0x80) or any character after it in the Macintosh character set, since they have their high bits set. This includes all the international characters, such as *â*, *é*, *î*, *ö*, and *ü*. For example, the function `putchar ('À')` does not write *À* but instead writes an inverted *K*.
- A non-printing character, such as Tab or Return, must have its high bit clear so the console can recognize and handle it correctly.
- Every time you call `cinverse()`, it erases all the console's text and moves the cursor to the beginning of the line it was in.

Using printf and scanf

4

Formatted input and output lets you completely control what your program's data looks like when you read it in or write it out. This chapter describes how to perform formatted input and output with these functions:

```
fprintf()    printf()     sprintf()
vfprintf()   vprintf()    vsprintf()
fscanf()     scanf()      sscanf()
```

The sections of this appendix fall into four parts:

- **Introduction**. The first section "Using Formatted I/O" describes how `printf()`, `scanf()`, and their related functions work.
- **How to Read and Write Data**. The next several sections describe how to make format specifiers to read and specific types of data.
- **Advanced Topics**. The next section "Advanced Features" describes how to take advantage of some advanced features of `printf()` and `scanf()`.
- **Summary**. The last two sections "printf Summary" and "scanf Summary" list all the options for each component of a format specifier.

Contents

Using Formatted I/O 51
Using formatted output 51
Using formatted input 52
Printing Integers 55
Printing plus signs 56
Printing pointers 57
Printing alternate forms of hexadecimal and octal integers 57
Specifying the number of digits to print 58
Specifying the total width of the number 58

Reading Integers . . . 60
Specifying the field size . . . 62
Letting the function decide the base . . . 62
Printing Floating-Point Numbers . . . 63
Printing in floating-point notation . . . 64
Printing in exponential notation . . . 65
Letting the function choose the notation . . . 65
Printing plus signs . . . 67
Specifying the total width of the number . . . 67
Printing Infinity and NaNs . . . 68
Reading Floating-Point Numbers . . . 69
Specifying the field size . . . 70
Printing Strings and Characters . . . 71
Printing Strings . . . 71
Printing characters . . . 73
Reading Strings and Characters . . . 73
Reading strings . . . 73
Reading characters . . . 75
Formatting Macintosh Toolbox Types . . . 75
Using OSType and ResType (four-byte constants) . . . 75
Using Pascal strings . . . 76
Advanced Features . . . 78
Getting the precision and field size from the argument list . . . 78
Suppressing assignment . . . 79
Finding the number of characters read or written . . . 80
Using scan sets . . . 82
printf Summary . . . 83
Flag character . . . 84
Field size . . . 85
Precision directive . . . 85
Argument size . . . 86
Conversion characters . . . 87
scanf Summary . . . 89
Flag character . . . 90
Field size . . . 90
Argument size . . . 90
Conversion characters . . . 91

Using Formatted I/O

`printf()`, `scanf()`, and their related functions let you specify exactly how the data you read or write looks like. This section describes how all these functions work.

Using formatted output

For more information on a function, see its description in Chapter 5, "The Standard Library Functions."

The `printf()` functions let you specify exactly how to format the data your print. These are the `printf()` functions:

`fprintf()`	`printf()`	`sprintf()`
`vfprintf()`	`vprintf()`	`vsprintf()`

These functions all print formatted data but to different places. `fprintf()` and `vfprintf()` print to a file, `printf()` and `vprintf()` print to the standard output stream, and `sprintf()` and `vsprintf()` print to a string. Since they're so closely related, this chapter uses `printf()` to refer to all of them.

`printf()` takes a format string and a list of arguments. It evaluates the arguments and prints them out according to the format string. The format string contains regular characters and format specifiers. `printf()` prints the regular characters. Each format specifier starts with a percent sign (`%`) and ends with a letter. It tells `printf()` how to print out one of the arguments. To print a percent sign (`%`), use `%%`.

For example, take this `printf()` statement:

```
printf( "c=%ld miles/second  pi=%5.3f\n",
        186000, 3.141569 );
```

It prints this line:

```
c=186000 miles/second  pi=3.142
```

The format string has two format specifiers: `%ld` and `%5.3f`. The specifier `%ld` tells `printf()` to print `186000` as a `long integer` number. The specifier `%5.3f` tells `printf()` to print `3.14159` as a floating-point number that takes up five characters and has three digits after the decimal point.

The `printf()` format specifier consists of five parts, labelled in Figure 4-1. This specifier prints a `long double` number in exponential notation.

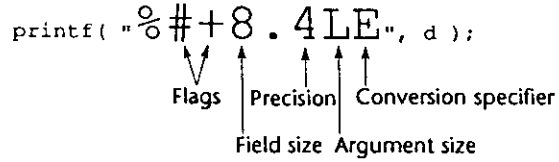

Figure 4-1 A format specifier for printf()

Here are short descriptions for each part of the format specifier. For more information on these parts, see section "printf Summary" on page 83.

- **The flags** modify the meaning of the specifier. You can have any number. In the example in Figure 4-1, the # flag specifies that a decimal point is always printed, and the + flag specifies that a plus sign is printed for positive integers.
- **The field size** is the minimum number of characters to print. The specifier fills in any extra space with spaces. The specifier in the example prints at least eight characters.
- **The precision** specifies how precisely to print out the argument's value. The specifier in the example prints four digits after the decimal point.
- **The argument size** is how much storage space the specifier's argument has. The argument for the specifier in the example is a `long double` number.
- **The conversion character** is the type of conversion to perform. The specifier in the example prints a floating-point value in exponential form (e.g., `1.86E+5`).

The only part of the specifier that's required is the conversion character. Any of the others can be left out.

Using formatted input

For more information on a function, see its description in Chapter 5, "The Standard Library Functions."

The `scanf()` functions convert a series of characters into a series of values, using a format string as its guide. These are the `scanf()` functions:

```
fscanf()        scanf()        sscanf()
```

These functions all read formatted data but to different places. `fscanf()` reads from a file, `scanf()` reads from the standard input stream, and

`sscanf()` reads from a string. Since they're so closely related, this chapter uses `scanf()` to refer to all of them.

`scanf()` takes a format string and a list of arguments. The arguments are pointers that tell `scanf()` where to put the values it reads. The format string contains regular characters, white space characters, and format specifiers:

- A format specifier starts with a percent sign (`%`) and ends with a letter. It tells `scanf()` how to convert a series of characters into a value.
- A white space characte is a space (`' '`), tab (`'\t'`), newline (`'\n'`), vertical tab (`'\v'`), return (`'\r'`), or formfeed (`'\f'`). It tells `scanf()` to skip ahead to the next character that's not white space and don't assign the white space characters to any argument.
- A regular character is any other character. Usually, the format string doesn't contain any. If it does, `scanf()` matches those characters with the characters it reads and doesn't assign them to its arguments.

For example, take this code fragment:

```
int i;
char s[5];

scanf( "%d %4s", &i, s );
printf( "You entered: %d and "%s"\n", i, s );
```

It prints this line:

```
32 good-bye
You entered: 32 and "good"
```

The `scanf()` format string has two format specifiers: `%d` and `%4s`. The specifier `%d` tells `scanf()` to read an integer. The specifier `%4s` tells `scanf()` to read a four-character-long string (and ignore any characters after the fourth).

The `scanf()` format specifier consists of four parts, as shown in Figure 4-1. The specifier in this example reads a six-digit `long integer` number and discards the value.

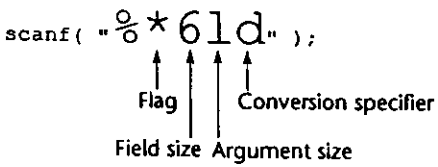

Figure 4-2 A format specifier for scanf()

Here are short descriptions for each part of the format specifier. For more information on these parts, see section "scanf Summary" on page 89.

- **The flag** modifies the meaning of the specifier. In this example, the flag * specifies that `scanf()` won't assign the value read to a variable.
- **The field size** is the maximum number of characters to read. The specifier in this example reads at most six characters.
- **The argument size** is how much storage space the specifier's argument has. The argument of the specifier in this example is a `long integer` number.
- **The conversion character** is the type of conversion to perform. The specifier in this example reads a decimal integer.

The only part of the specifier that's required is the conversion character. Any of the others can be left out.

Printing Integers

`printf()` lets you print integers in a wide variety of formats. This table includes most of the format specifiers you'll need:

You can also print integers as characters. For more information, see section "Printing characters" on page 73.

To print an integer of this type...	In this base...	Use this specifier...
`int`	decimal	`%d`
`long int`	decimal	`%ld`
`short int`	decimal	`%hd`
`unsigned int`	decimal	`%u`
`unsigned int`	octal	`%o`
`unsigned int`	hexadecimal	`%X` or `%x`
`unsigned long int`	decimal	`%lu`
`unsigned long int`	octal	`%lo`
`unsigned long int`	hexadecimal	`%lX` or `%lx`
`unsigned short int`	decimal	`%hu`
`unsigned short int`	octal	`%ho`
`unsigned short int`	hexadecimal	`%hX` or `%hx`

Keep these facts in mind:

- The octal and hexadecimal specifiers interpret their arguments as unsigned integers. If their arguments are negative numbers, these specifiers will not print what you expect.
- `%i` is an alternate for `%d`, and `%li` is an alternate for `%ld`. `%i` and `%li` format integers exactly like `%d` and `%ld`, but `%d` and `%ld` are more commonly used
- You have two choices when you print integers in hexadecimal. The specifiers `%X` and `%lX` print the hexadecimal digits in uppercase (`A`, `B`, `C`, `D`, `E`, and `F`). The specifiers `%x` and `%lx` print the hexadecimal digits in lowercase (`a`, `b`, `c`, `d`, `e`, and `f`).

This program demonstrates the format specifiers:

```
#include <stdio.h>

main()
{
   int i = 456;
   long int l = 123123123L;
   unsigned int ui = 45645U;
   unsigned long int ul = 2123123123UL;

   printf( "In decimal:\n" );
   printf( "   i = %d\n", i );
```

```
    printf( "   l = %ld\n", l );
    printf( "  ui = %u\n", ui );
    printf( "  ul = %lu\n\n", ul );

    printf( "In octal:\n" );
    printf( "  ui = %o\n", ui );
    printf( "  ul = %lo\n\n", ul );

    printf( "In hexadecimal:\n" );
    printf( "  ui = %X\n", ui );
    printf( "  ul = %lX\n", ul );
}
```

This is what the program prints:

```
In decimal:
   i = 456
   l = 123123123
  ui = 45645
  ul = 2123123123

In octal:
  ui = 131115
  ul = 17643044663

In hexadecimal:
  ui = B24D
  ul = 7E8C49B3
```

The rest of this section describes more sophisticated ways to format integers.

Printing plus signs

Usually, `printf()` prints a sign only for negative integers, so columns of positive and negative numbers might not line up. However, you can force `printf()` to print something in the front of positive numbers. To print a positive integer with a plus sign, use the + flag. To print a positive integer with a space in front, use the space flag. For example, these statements:

```
printf( "|% d|  |%+d|  |%d|\n",
        -123, -123, -123 );
printf( "|% d|  |%+d|  |%d|\n",
        123, 123, 123 );
```

print this:

```
|-123|  |-123|  |-123|
| 123|  |+123|  |123|
```


Printing pointers

To print the value of a pointer, use the specifier `%p`. It prints the address in hexadecimal, with exactly eight digits. For example, this program prints the address of the string `s`:

```
#include <stdio.h>

main()
{
    char s[] = "Hello World";
    printf( "The address of s is %p.\n", s );
}
```

This is sample output from the program:

```
The address of s is 0019B6F0.
```

Printing alternate forms of hexadecimal and octal integers

The specifiers `%x` and `%o` (and their variants) print numbers that are easily confused with decimal integers. To print hexadecimal and octal integers that stand out more, use the # flag. The # flag prefixes the numbers, according to this table:

These specifiers...	prefix numbers with...
`%#x` and `%#lx`	0x
`%#X` and `%#lX`	0X
`%#o` and `%#lo`	0

For example, these statements:

```
printf( "|%x|\t|%#x|\n", 123, 123 );
printf( "|%X|\t|%#X|\n", 123, 123 );
printf( "|%o|\t|%#o|\n", 123, 123 );
```

print this:

```
|7b|    |0x7b|
|7B|    |0X7B|
|173|   |0173|
```

Note

To prefix an octal number with a zero, the specifiers `%#o` and `%#lo` increase the number's precision to force the first digit to be a zero. For more information on the precision, see section "Specifying the number of digits to print" on page 58.

Specifying the number of digits to print

For integers, the precision specifies the minimum number of digits to print. `printf()` pads the integer with zeroes until the number of digits reaches the precision. It adds the zeros after the sign or `0x`, but before the digits. If an integer has more digits than the precision specifies, `printf()` prints the extra digits.

These statements print 123 in three different ways, but always with four digits:

```
printf( "|%.4d|\n", 123 );
printf( "|%+.4d|\n", 123 );
printf( "|%#.4X|\n", 123 ); /* in hexadecimal */
```

The statements print this:

```
|0123|
|+0123|
|0X007B|
```

Ordinarily, if the precision is zero and the integer to print is zero, `printf()` prints nothing. But if you use the space flag, `printf()` prints a space. For example, these statements:

You can read the precision from `printf()`'s argument list. For more information, see section "Getting the precision and field size from the argument list" on page 78

```
printf( "|%.0d|\n", 0 );
printf( "|% .0d|\n", 0 );
```

print this:

```
||
| |
```

Specifying the total width of the number

The field size specifies the minimum number of spaces the integer takes up, including digits, a plus or minus sign, and `0x` or `0X` (an option for hexadecimal numbers).If an integer takes up fewer characters than the field size, `printf()` pads the number with spaces. If an integer takes up more characters, `printf()` prints the additional characters.

If the field size is positive, `printf()` right-justifies the number by adding spaces before it (and before any sign or `0x`). If the field size is negative, `printf()` left-justifies the number by adding spaces after it.

These statements print 123 in six different ways, but it always takes up five spaces: (The first column is right-justified, the second column is left-justified.)

```
printf( "|%5d| |%-5d|\n", 123, 123);
printf( "|%+5d| |%+-5d|\n", 123, 123);
printf( "|%#5X| |%#-5X|\n", 123, 123);
```

The statements print this:

```
|  123| |123  |
| +123| |+123 |
| 0X7B| |0X7B |
```

The field size lets you pad with zeros instead of spaces. If you use the `0` (zero) flag character, `printf()` pads the number with zeroes until it reaches the field size. It adds the zeros after the sign or `0x`, but before the digits. For example, these statements print 123 in three different ways, padding with zeroes:

```
printf( "|%05d|\n", 123);
printf( "|%+05d|\n", 123);
printf( "|%#05X|\n", 123);
```

You can read the field size from `printf()`'s argument list. For more information, see section "Getting the precision and field size from the argument list" on page 78

The statements print this:

```
|00123|
|+0123|
|0X07B|
```

Notice that this method doesn't guarantee that the integer always has the same number of digits, which is usually what you want when you pad with zeros. To control the number of digits, use the precision directive, described in section "Specifying the number of digits to print" on page 58.

This is how you might use the field size and precision together:

```
printf( "|%+6.4d|\n", 123 ); /* in decimal     */
printf( "|%6.4o|\n", 123 );  /* in octal       */
printf( "|%#6.4X|\n", 123 ); /* in hexadecimal*/
```

The statements print this:

```
| +0123|
|  0173|
|0X007B|
```

Reading Integers

`scanf()` lets you read integers in a wide variety of formats. This table describes most of the format specifiers you'll need:

To read an integer of this type...	of this base...	Use this specifier...
`int`	decimal	`%d`
`long int`	decimal	`%ld`
`short int`	decimal	`%hd`
`unsigned int`	decimal	`%u`
`unsigned int`	octal	`%o`
`unsigned int`	hexadecimal	`%X` or `%x`
`unsigned long int`	decimal	`%lu`
`unsigned long int`	octal	`%lo`
`unsigned long int`	hexadecimal	`%lX` or `%lx`
`unsigned short int`	decimal	`%hu`
`unsigned short int`	octal	`%ho`
`unsigned short int`	hexadecimal	`%hX` or `%hx`
`int`	octal, decimal, or hexadecimal	`%i`
`long int`	octal, decimal, or hexadecimal	`%li`
`short int`	octal, decimal, or hexadecimal	`%hi`

Notice that `scanf()` doesn't treat the specifiers `%x` and `%X` (or `%lx` and `%lX`) differently. For example, both `%x` and `%X` match `0xA1` and `0XaI`.

`scanf()` skips over any whitespace and looks for an integer formatted according to this table:

If you're reading a number in base...	The function looks for an integer formatted like...
decimal	`[+|-] {`*digits*`}`
octal	`[+|-] [0] [`*digits*`]`
hexadecimal	`[+|-] [0x|0X] [`*digits*`]`

Anything in brackets (`[]`) is optional. If items are separated by vertical bars (`|`), only one of those items may be in that position. Remember that the hexadecimal digits are the numerals `0-9` and the letters `A-F` or `a-f`, and the octal digits are the numerals `0-7`.

Keep these facts in mind:

- Since an unsigned number can't be negative, the specifiers for unsigned numbers (`%o`, `%lo`, `%u`, `%lu`, `%x`, `%X`, `%lx`, and `%lX`) won't match a numbers that start with minus signs.
- The specifiers `%i` and `%li` figure out the base of a number by looking at its prefix. For more information, see section "Letting the function decide the base" on page 62.

This example reads three decimal numbers, three octal numbers and three hexadecimal numbers:

```
#include <stdio.h>

main()
{
   int i1, i2, i3;

   printf( "Enter three decimal numbers: " );
   scanf( "%d%d%d", &i1, &i2, &i3 );
   printf( "You entered: %d\t%d\t%d\n",
           i1, i2, i3 );

   printf( "Enter three octal numbers: " );
   scanf( "%o%o%o", &i1, &i2, &i3 );
   printf( "You entered: %d\t%d\t%d\n",
           i1, i2, i3 );

   printf( "Enter three hexadecimal numbers: ");
   scanf( "%x%x%x", &i1, &i2, &i3 );
   printf( "You entered: %d\t%d\t%d\n",
           i1, i2, i3 );
```

This is sample output from the program:

```
Enter three decimal numbers: 1234 -987 0
You entered: 1234        -987    0
Enter three octal numbers: 1234 034 198
You entered: 668         28      1
Enter three hexadecimal numbers: 1234 0XA1 cab
You entered: 4660        161     3243
```

Notice that the specifier `%o` ignored the digits `9` and `8` in the number `198`. Since `9` and `8` aren't valid octal digits, the specifier `%o` stopped reading when it reached them.

Specifying the field size

To read in number that takes up a certain number of characters, include a field size in the specifier. `scanf()` reads in only that number of characters. For example, this program reads in three numbers that at most take up 4 characters:

```
#include <stdio.h>

main()
{
   int i1, i2, i3;

   scanf( "%4d%4d%4d", &i1, &i2, &i3 );
   printf( "You entered: %d\t%d\t%d\n",
           i1, i2, i3 );
}
```

This is sample output from the program:

```
1234567 123
You entered: 1234        567      123
```

Notice that `scanf()` interpreted `12345678` as two numbers: `1234` and `567`.

Letting the function decide the base

To let `scanf()` decide what base a number is in, use the specifiers `%i` or `%li`. To determine the base, `scanf()` looks at the number's prefix:

If the prefix is...	the base is...
0x or 0X	16 (hexadecimal)
0 (zero)	8 (octal)
none	10 (decimal)

For example, this code fragment reads six integers and prints them out:

```
int i1, i2, i3, i4, i5, i6;
scanf( "%i%i%i%i%i%i",
       &i1, &i2, &i3, &i4, &i5, &i6 );
printf( "%d\t%d\n%d\t%d\n%d\t%d\n",
        i1, i2, i3, i4, i5, i6 );
```

If you execute the statements and enter this:

```
0X10  0xFACE  01056  0123  10  123
```


they print this:

```
16      32767
558     83
10      123
```

The program interprets `0X10` and `0xFACE` as hexadecimal, `01056` and `0123` as octal, and `10` and `123` as decimal.

Printing Floating-Point Numbers

`printf()` lets you print floating-point numbers in either floating-point or exponential notation. Floating-point notation is for numbers of moderate value and looks like `3.141569` or `-123.456`. Exponential notation is for very large or very small numbers and looks like `1.23E45` or `8.9E-76`.

This table includes most of the format specifiers you'll need:

To print a number of this type...	In this format...	Use this specifier...
`float`, `double`, or `short double`	floating-point	`%f`
`float`, `double`, or `short double`	exponential	`%e` or `%E`
`float`, `double`, or `short double`	floating-point or exponential	`%g` or `%G`
`long double`	floating-point	`%Lf`
`long double`	exponential	`%Le` or `%LE`
`long double`	floating-point or exponential	`%Lg` or `%LG`

Keep these pointers in mind:

- To print `long double` numbers, use a capital `L` in the specifier, not a lower-case `l`. The lower-case `l` is for printing `long integer` numbers.
- The specifiers `%g`, `%G`, `%Lg`, and `%LG` can print out a number in either floating-point or exponential format, depending on the number of significant digits. For more information, see section "Letting the function choose the notation" on page 65.

This program demonstrates how the floating-point format specifiers work:

```
#include <stdio.h>

main()
{
   double d = 123.45;
   long double ld = 9.87e35;

   printf( "|%G|\t|%E|\t|%f|\n\n", d, d, d );
   printf( "|%LG|\t|%LE|\n|%Lf|\n", ld,ld,ld );
}
```

The program prints this:

```
|123.45|        |1.234500E+02|   |123.4500001

|9.87E+35|      |9.870000E+35|
|9869999999999999999000000000000000000.000000|
```

Printing in floating-point notation

To print floating-point numbers in floating-point notation, use the specifier `%f`. It prints numbers in this style:

You can read the precision from `printf()`'s argument list. For more information, see section "Getting the precision and field size from the argument list" on page 78

```
[-] ddd. dddddd
```

The precision specifies how many digits appear after the decimal point. If the precision isn't given, `printf()` uses 6. If the precision is zero, `printf()` doesn't print a decimal point. For example, these statements print 3.14157 three different ways:

```
printf( "|%f|\n", 3.14157 );
printf( "|%.3f|\n", 3.14157 );
printf( "|%.0f|\n", 3.14157 );
```

The statements print this:

```
|3.141570|
|3.142|
|3|
```

To force `printf()` to print a decimal point even when the precision is zero, use the # flag. For example, this statement:

```
printf( "|%#.0f|", 3.14157 );
```

prints this:

```
|3.|
```


Printing in exponential notation

To print a floating-point number in exponential notation, use the specifier `%e` or `%E`. They print numbers in this style:

You can read the precision from `printf()`'s argument list. For more information, see section "Getting the precision and field size from the argument list" on page 78

```
[-] d. dddddde± dd    (For %e)
[-] d. ddddddE± dd    (For %E)
```

The precision specifies how many digits appear after the decimal point. If the precision isn't given, `printf()` uses 6. If the precision is zero, `printf()` doesn't print a decimal point. For example, these statements print 1.23456×10^{123} three different ways:

```
printf( "|%E|\n", 1.23456e123 );
printf( "|%.3E|\n", 1.23456e123 );
printf( "|%.0E|\n", 1.23456e123 );
```

The statements print this:

```
|1.234560E+123|
|1.235E+123|
|1E+123|
```

There is always one digit before the decimal point. That digit is zero only if the number is zero. The exponent always has at least two digits. If the number is zero, the exponent is zero. For example, this statement:

```
printf( "|%E|\n", 0.0 );
```

prints this:

```
|0.000000E+00|
```

To force `printf()` to print a decimal point even when the precision is zero, use the # flag. For example, this statement:

```
printf( "|%#.0E|\n", 1.23456e123 );
```

prints this:

```
|1.E+123|
```

Letting the function choose the notation

To let `printf()` choose the best way to print a number, use the specifier `%g` or `%G`. These specifiers choose between the floating-point or exponential notation. (When printing in exponential notation, `%g` prints the *e* before the exponent in lowercase, and `%G` prints the *E* in uppercase.)

The specifiers %g and %G differ from %f, %e, and %E in a few ways. For %g and %G, printf() doesn't print trailing zeros after the decimal point. For example, see how %f, %E and %G handle the same numbers:

```
printf( "|%f|\t|%G|\n", 3.1, 3.1 );
printf( "|%E|\t|%G|\n", 3.1e10, 3.1e10 );
```

The statements print this:

```
|3.100000|       |3.1|
|3.100000E+10|   |3.1E+10|
```

You can read the precision from printf()'s argument list. For more information, see section "Getting the precision and field size from the argument list" on page 78

Also, %g and %G interpret the precision differently. The precision specifies the maximum number of significant digits to print. The significant digits include the digits both before and after the decimal point. If the precision isn't given, printf() uses 6. If the precision is zero, printf() uses 1 instead. For example, notice that this example uses different values for the precision for %G to get the same results it gets with %f and %E:

```
printf( "|%.4f|\t|%.5G|\n", 9.8765, 9.8765 );
printf( "|%.2E|\t|%.3G|\n", 123e45, 123e45 );
```

The statements print this:

```
|9.8765|          |9.8765|
|1.23E+47|        |1.23E+47|
```

These specifiers choose the notation depending on the value of the number. They choose the exponential notation if the exponent of the number would be less than -4 or greater than or equal to the precision. For example, see how these statements print their numbers:

```
printf( "|%G|\t|%G|\n", -0.0001, -0.00001 );

/* Remember that the default precision is 6. */
/* Note that it doesn't print trailing zeroes */
printf( "|%G|\t|%G|\n", 3.5e5, 3.5e6 );

printf( "|%.11G|\t|%.11G|\n", 1.2e10, 1.2e11 );
```

The statements print this:

```
|-0.0001|          |-1E-05|
|350000|           |3.5E+06|
|12000000000|      |1.2E+11|
```


To force printf() to print trailing zeroes, use the # flag. For example, these statements:

```
printf( "|%#G|\t|%G|\n", 3.1, 3.1 );
printf( "|%#G|\t|%G|\n", 3.1e10, 3.1e10 );
```

print this:

```
|3.10000|          |3.1|
|3.10000E+10|     |3.1E+10|
```

Printing plus signs

Usually, printf() prints a sign only for negative numbers, so columns of positive and negative numbers might not line up. However, you can force printf() to print something in the front of positive numbers. To print a positive number with a plus sign, use the + flag. To print a positive number with a space in front, use the space flag. For example, these statements:

```
printf( "|% .1f|   |% .3E|\n", 123.4, 123.4 );
printf( "|%+.1f|   |%+.3E|\n", 123.4, 123.4 );
printf( "|% .1f|   |%+.3E|\n", -123.4, -123.4 );
```

print this:

```
| 123.4|    | 1.234E+02|
|+123.4|    |+1.234E+02|
|-123.4|    |-1.234E+02|
```

Specifying the total width of the number

The field size specifies the minimum number of spaces the number takes up, including digits, the exponent, a plus or minus sign. If a number takes up fewer characters than the field size, printf() pads it with spaces. If a number takes up more characters, printf() prints the additional characters.

If the field size is positive, printf() right-justifies the number by adding spaces before it (and before any sign). If the field size is negative, printf() left-justifies the number by adding spaces after it.

These statements print 9.8765 and 1.23×10^{45} in four different ways, but they always take fifteen spaces: (The first column is right-justified, the second column is left-justified.)

```
printf( "|%15f|\t|%-15f|\n", 9.8765, 9.8765 );
printf( "|%15G|\t|%-15G|\n", 9.8765, 9.8765 );
printf( "|%15E|\t|%-15E|\n", 123e45, 123e45 );
printf( "|%15G|\t|%-15G|\n", 123e45, 123e45 );
```

The statements print this:

```
|        9.876500|        |9.876500        |
|          9.8765|        |9.8765          |
|  1.230000E+47|        |1.230000E+47    |
|       1.23E+47|        |1.23E+47        |
```

The field size lets you to pad a floating-point number with zeros. If you use the 0 (zero) flag character, `printf()` pads the number with zeroes until it reaches the field size. It adds the zeros after the sign, but before the digits. For example, these statements print 9.8765 and 1.23 x 10^{45} in two different ways, padding with zeroes:

```
printf( "|%015f|\n", 9.8765 );
printf( "|%015G|\n", 9.8765 );
printf( "|%015E|\n", 123e45 );
printf( "|%015G|\n", 123e45 );
```

The statements print this:

```
|00000009.876500|
|0000000009.8765|
|0001.230000E+47|
|00000001.23E+47|
```

You can use the field size and precision together to fine tune how your numbers appear. This is an example:

```
printf( "|%5.3f|\n", 9.8765 );
printf( "|%5.4G|\n", 9.8765 );
printf( "|%7.2E|\n", 123e45 );
printf( "|%7.3G|\n", 123e45 );
```

You can read the field size from `printf()`'s argument list. For more information, see section "Getting the precision and field size from the argument list" on page 78

The statements print this:

```
|9.877|
|9.877|
|1.23E+47|
|1.23E+47|
```

Printing Infinity and NaNs

Any floating-point specifier can print a number with the value Infinity or NaN (Not a Number). It prints Infinity as `INF` and NaN as `NAN`*dd*, where *dd* is a hexadecimal number representing the type of NaN. For example, this program prints positive and negative Infinity and an NaN:

Some math functions in the ANSI library return Infinity when they produce a value too large to represent. The ANSI library doesn't use NaN, but the SANE library does. For more information on SANE, see the book Apple Numerics Manual, Second Edition *(Addison-Wesley).*

```
#include <stdio.h>
#include <math.h>
#include <SANE.h>

main()
{
    double notnum = nan( -1 );
    double posinf = cosh( 1e10 );
    double neginf = log( 0 );

    printf( " nan = %G\n", notnum );
    printf( "+inf = %G\n", posinf );
    printf( "-inf = %G\n", neginf );
}
```

The program prints this:

```
 nan = NANFF
+inf = INF
-inf = -INF
```

You can't read Infinity or NaN values with `scanf()` or any other function in the ANSI library.

Reading Floating-Point Numbers

`scanf()` lets you read floating-point numbers in a wide variety of formats. This table describes most of the format specifiers you'll need:

To read a number of this type...	Use one of these specifiers...
`float`	`%e, %E, %f, %g, %G`
`double`	`%le, %lE, %lf, %lg, %lG`
`short double`	`%he, %hE, %hf, %hg, %hG`
`long double`	`%Le, %LE, %Lf, %Lg, %LG`

Notice how `scanf()` interprets these specifiers differently from `printf()`:

- All the specifiers can match numbers in either floating-point and exponential notation. For example, `%f` and `%e` can match `1.23`, `9.8e10`, and `1.0E-10`.
- You need to use different specifiers for `double` and `float` numbers. To read a `double` number, you need to include a low-

er-case l in the specifier. To read a `long double` number, you still need to include an upper-case L.

`scanf()` skips over any whitespace and looks for a floating-point number formatted like this:

`[+|-] [`*digits*`] [.` *digits*`] [e|E [-|+]` *digits*`]`

Anything in brackets (`[]`) is optional. If items are separated by vertical bars (`|`), only one of those items may be in that position.

For example, this program reads six floating-point numbers and writes them out:

```
double f1, f2, f3, f4, f5, f6;
scanf( "%lf%lf%lf%lf%lf%lf",
       &f1, &f2, &f3, &f4, &f5, &f6 );
printf( "f1=%G\tf2=%G\tf3=%G\n"
        "f4=%G\tf5=%G\tf6=%G\n"
        f1, f2, f3, f4, f5, f6 );
```

If you ran the program and enter this line:

```
1.0e32  -325E+1  2.3E-10  987.1  1000000  .456
```

the program prints this:

```
f1=1E+32        f2=-3250        f3=2.3E-10
f4=987.1        f5=1E+06        f6=0.456
```

Specifying the field size

To read in number that takes up a certain number of characters, include a field size in the specifier. `scanf()` reads in only that number of characters. For example, this program reads in three numbers that at most take up five characters:

```
#include <stdio.h>

main()
{
   double d1, d2, d3;

   scanf( "%5lf%5lf%5lf", &d1, &d2, &d3 );
   printf( "You entered: %G\t%G\t%G\n",
           d1, d2, d3 );
}
```


This is sample output from the program:

```
0.1234567 2.0e34
You entered: 0.123      4567    2000
```

Notice that `scanf()` read in `0.1234567` as two numbers (`0.123` and `4567`) and read in `2.0e34` as `2.0e3` (or `2000`).

Printing Strings and Characters

To print characters, use the specifiers `%s` and `%c`. The specifier `%s` prints a string of characters. The specifier `%c` prints one character.

Printing Strings

The simplest way to print a string is to include it in the format specifier. For example, this statement prints "Hello world."

```
printf( "Hello world.\n" );
```

To print a percent sign (`%`), use `%%`. For example, this statement prints "100%."

```
printf( "100%%" );
```

To print a string from an argument, use the `%s` specifier. For example, this code fragment:

```
char s[] = "world";
printf( "hello, %s.\n", s );
```

prints this:

```
hello, world.
```

You can read the precision or field size from `printf()`'s argument list. For more information, see section "Getting the precision and field size from the argument list" on page 78

To specify the minimum number of characters the printed string takes up, use the field size. If the field size is positive, `printf()` adds spaces to the left of the string. If the field size is negative, `printf()` adds spaces to the right of the string. For example, this statement:

```
printf( "|%10s|\t|%-10s|\n", "hello", "hello" );
```

prints this:

```
|     hello|     |hello     |
```

Warning
To print a string that's not null-terminated, you must include a precision. Otherwise, your program may crash as it searches for the final null character (`'\0'`)

To specify the maximum number of characters the printed string takes up, use the precision. For example, this statement:

```
printf( "|%.5s|\n", "good-bye" );
```

prints this:

```
|good-|
```

To specify a range of acceptable sizes, use the field size and precision together. For example, this code fragment prints a few strings, making sure they take up between 4 to 6 characters:

```
printf( "|%4.6s|\n", "123456" );
printf( "|%4.6s|\n", "hi" );
printf( "|%4.6s|\n", "hello" );
printf( "|%4.6s|\n", "good morning" );
```

The example prints this:

```
|123456|
|  hi|
|hello|
|good m|
```

And this example prints a left-justified list of names, making sure each takes up exactly 15 characters:

```
printf( "|%-15.15s|\n","Smith, John" );
printf( "|%-15.15s|\n","Capraro, Marie" );
printf( "|%-15.15s|\n","Jones, Christopher" );
```

The example prints this:

```
|Smith, John     |
|Capraro, Marie  |
|Jones, Christop|
```


Printing characters

To print an object of type `char`, `integer`, `unsigned char`, or `unsigned integer` as a character, use the `%c` specifier. For example, this code fragment:

```
char c = 'c';
unsigned int i = 0x63;

printf( "|%c|\t|%c|\n", c, i );
```

prints this:

```
|c|     |c|
```

If you give the %c specifier an unprintable character, it prints nothing. For example, this code fragment prints nothing for `'\1'` but it prints a tab for `'\t'`.

```
char c1 = '\1';
char c2 = '\t';
printf ("c1='%c'\tc2='%c'\n", c1, c2);
```

The code fragment prints this:

```
c1=''   c2='    '
```

To pad the character with spaces, either add the spaces to the format string yourself or use the field size. If the field size is positive, `printf()` adds the spaces to the left of the character. If the field size is positive, `printf()` adds the spaces to the right of the character. For example, this statement:

```
printf( "|%5c|\t|%-5c|\n", 'a', 'a' );
```

prints this:

```
|    a| |a    |
```

The `%c` specifier ignores the precision.

Reading Strings and Characters

To read strings or characters, use the specifiers `%s` and `%c`. The specifier `%s` reads a string of characters with no whitespace. The specifier `%c` reads a specific number of characters, including whitespace.

Reading strings

The specifier `%s` skips over any whitespace characters, matches a sequence of characters, and stops when it reaches a whitespace character. It appends a

null character (`'\0'`) to its argument when its done. For example, this code fragment reads in three strings and then writes them out.

```
char s1[10], s2[10], s3[10];
scanf( "%s%s%s", s1, s2, s3 );
printf( "s1=|%s|\ts2=|%s|\ts3=|%s|\n",
        s1, s2, s3 );
```

If you run the program and enter this:

```
hello there, world.
```

the program writes this:

```
s1=|hello|      s2=|there,|      s3=|world.|
```

Note

Make sure that your variable is large enough to hold the string read in and the terminating null character (`'\0'`). If the variable is too small, you will write over other items in memory.

If you include a field size in the specifier, `scanf()` matches at most that many characters, and then appends a null character to its argument. For example, this code reads in strings that are at most 4 characters long and then writes them out:

```
char s1[10], s2[10], s3[10];
scanf( "%4s %4s %4s", s1, s2, s3 );
printf( "s1=|%s|\ts2=|%s|\ts3=|%s|\n",
        s1, s2, s3 );
```

If you run the program and enter this:

```
Hi there, world.
```

the program writes this:

```
s1=|Hi| s2=|ther|       s3=|e,|
```


Reading characters

The specifier `%c` matches a character, even if it's a whitespace character. For example, this code fragment reads in three characters and writes them out:

```
char c1, c2, c3;
scanf( "%c%c%c", &c1, &c2, &c3 );
printf( "c1='%c'\tc2='%c'\tc3='%c'\n",
        c1, c2, c3 );
```

If you run the program and enter this:

```
a b
```

the program writes this:

```
c1='a'  c2=' '  c3='b'
```

If you include a field size in the specifier, `scanf()` matches that many characters, including whitespaces. It does not append a null character (`'\0'`) to its argument. For example, this code reads in 3 sets of 4 characters and then writes them out:

```
char c1[4], c2[4], c3[4];
scanf( "%4c%4c%4c", c1, c2, c3 );
printf( "c1='%.4s'\tc2='%.4s'\tc3='%.4s'\n",
        c1, c2, c3 );
```

If you run the program and enter this:

```
Hi there, world.
```

the program writes this:

```
c1='Hi t'       c2='here'       c3=', wo'
```

Formatting Macintosh Toolbox Types

The Macintosh Toolbox uses two types that can be especially difficult to format: Pascal strings and four-character strings stored as `long integer` numbers. This section explains how to read and write them.

Using OSType and ResType (four-byte constants)

The Macintosh Toolbox sometimes stores four-character strings as long integers, like `'ICON'`, `'TEXT'`, or `'MACA'`. These strings include resource types (`ResType`), file types (`OSType`), and file creator signatures (`OSType`).

To print a long integer as a string, use `%.4s` (note the period) as the format specifier and pass the address of the number. For example, this program prints out the file type and creator for a THINK C source file.

```
#include <stdio.h>
#include <FileMgr.h>

main()
{
   FInfo fileInfo;

   GetFInfo( "\pMyFile.c", 0, &fileInfo );

   printf( "File Type: '%.4s'  "
           "File Creator: '%.4s'\n",
           &fileInfo.fdType,
           &fileInfo.fdCreator );
}
```

The program prints this line:

```
File Type: 'TEXT'  File Creator: 'KAHL'
```

To read a four-character string into a long integer, use `%4c` as the format specifier and pass the address of the number. For example, this program reads a file type and prints it out, as both an integer and a string:

```
#include <stdio.h>

main()
{
   OSType type;

   printf( "Enter a file type: " );

   scanf( "%4c", &type );

   printf( "You entered %lu or '%.4s'.\n",
           (unsigned long) type, &type );
}
```

This is sample output from the program:

```
Enter a file type: TEXT
You entered 1413830740 or 'TEXT'.
```

Using Pascal strings

The Macintosh Toolbox uses Pascal strings, which begin with a length byte, instead of null-terminated C strings. To print Pascal strings, use `%#s` as the format specifier and pass the string. For example, this program prints the string `CurApName`, a Toolbox global that contains the name of the currently running application.


```
#include <stdio.h>

main()
{
   printf( "CurApName is \"%#s\".\n",
           CurApName );
}
```

If you named this program `Sample Project` and ran it under THINK C, it would print this line:

```
CurApName is "Sample Project".
```

Note

The `%#s` format specifier is a THINK C extension that's not part of the ANSI standard.

You can't read in a string as a Pascal string. However, you can read in a null-terminated string and immediately convert it with `CtoPstr()`. For example, this program asks you to enter a filename, and renames `MyFile` to what you entered.

```
#include <stdio.h>

main()
{
   char newName[80];

   printf( "Enter a new name: " );

   scanf( "%s", newName );
   CtoPstr( newName );

   /* MyFile must be in the same folder */
   /* as this program.                  */
   Rename( "\pMyFile", 0, newName );
   printf( "MyFile is now %#s.\n", newName );
}
```

This is sample output from the program:

```
Enter a new name: YourFile
MyFile is now YourFile.
```

Advanced Features

The section describes how to use some advanced features of `printf()` and `scanf()`:

- Getting the precision and field size from the argument list with `printf()`
- Suppressing assignment with `scanf()`
- Finding the number of characters `printf()` or `scanf()` reads or writes.
- Using scan sets with `scanf()`.

Getting the precision and field size from the argument list

Sometimes, you need to compute the proper precision and field size values for a specifier and you can't include them directly in your code. In these cases, use an asterisk for the field size or precision, and `printf()` uses the next argument as the value. For example, these statements print pi with a field size of eight and a precision of four: (Notice that the field size and precision come before pi in the argument list.)

```
int fs = 8;
int p = 4;
double pi = 3.141569;

printf( "pi = %*.*d\n", fs, p, pi );
```

The statements print this:

```
pi =   3.1416
```

The asterisk is especially useful when you're printing a table and you don't know in advance the size of the columns. For example, this program prints a small table with a width you specify.

```
#include <stdio.h>
struct person {
   char name[80];
   int age;
};

main()
{
   int tableWidth, nameWidth, i;
   int ageWidth = 3;
   struct person men[3] =
      { { "Johnson, Frederick", 25 },
        { "Smith, John", 42 },
        { "Goldenstein, Lawrence", 38 } };
```



```
   printf( "Enter the width for the table: " );
   scanf( "%d", &tableWidth );

   nameWidth = tableWidth - ageWidth - 3;

   printf( "\n|%*.*s|%*.*s|\n",
           -nameWidth, nameWidth, "NAME",
           -ageWidth, ageWidth, "AGE" );

   for ( i=0; i<3; i++ )
      printf( "|%*.*s|%*d|\n",
              -nameWidth,nameWidth,men[i].name,
              ageWidth, men[i].age );
}
```

This is sample output from the program:

```
Enter the width for the table: 25

|NAME                     |AGE|
|Johnson, Frederick       | 25|
|Smith, John              | 42|
|Goldenstein, Lawren      | 38|
```

Suppressing assignment

Sometimes you need to scan a line that contains text you don't need. If you know exactly what the text is, include it directly in the format string. For example, say you're reading a file of numbers separated by commas. This code fragment reads three numbers and prints them out.

```
scanf( "%lf, %lf, %lf", &d1, &d2, &d3 );
printf( "You read: %G\t%G\t%G\n", d1, d2, d3 );
```

If the file contains this line:

```
123e12, 3.21, 89e67
```

the example prints this:

```
You read: 1.23E+14      3.21    8.9E+68
```

To read a percent sign, use `%%`. For example, this code fragment reads a percentage.

```
scanf( "%d%%", &percentage );
printf( "That's %d%%", percentage );
```

If the file contains this line:

```
100%
```

the example prints this:

```
That's 100%
```

If you don't know what the text is, use a format specifier with the assignment suppressing flag *. The specifier matches the input, but doesn't assign it to an argument. For example, say you have a text file of employee records that contain an employee's ID number, name, and salary. These statements read the ID number and salary — but not the name — and then prints them out:

```
fscanf( fp, "%4d%*s%*s%ld", &id, &salary );
printf( "Employee #%.4d earns $%ld a year.\n",
        id, salary);
```

If the file contains this line:

```
0134 Mary Wu 40000
```

the example prints this:

```
Employee #0134 earns $40000 a year.
```

Finding the number of characters read or written

Both `printf()` and `scanf()` let you find out how many characters the function has read or written so far. You have two ways:

- Use the `%n` specifier.
- Use the function's return value.

To use the `%n` specifier, put it in your format string and pass a pointer to an integer. The function (either a `scanf()` *or* `printf()` function) puts the number of characters processed so far in the integer. For example, this code fragment asks you for two numbers and prints out how many characters you used to enter each. It then tells you how many characters it need to print out that information:

```
int n1, n2, n3;
double d1, d2;

printf( "Enter two numbers: " );
scanf( "%lf%n%lf%n", &d1, &n1, &d2, &n2 );
printf( "%G was %d characters long.\n"
        "%G was %d characters long.\n%n",
        d1, n1, d2, n2-n1-1, &n3 );
printf( "And those last two lines are "
        "%d characters long.\n",
        n3 );
```

This is sample output:

```
Enter two numbers: 1.2345 9.8765
1.2345 was 6 characters long.
9.8765 was 6 characters long.
And those last two lines are 60 characters long.
```

The return value for `printf()` is the total number of characters written or read. For example, this code fragment uses the return value to find the number of characters that a `printf()` statement writes:

```
int num;
num = printf( "Hello world.\n" );
printf( "That took %d characters.\n", num );
```

That code fragment writes this:

```
Hello world.
That took 13 characters.
```

The return value for `scanf()` is the number of items it read and assigned to variables. This total doesn't include suppressed conversions or the `%n` speci-

fier For example, this code fragment prints the number of characters and items you enter.

```
#include <stdio.h>

main()
{
   int a, b, c, d, chars=0, items;
   printf( "Enter four integers: ");
   items = scanf( "%d%d%d%d%n",
                  &a, &b, &c, &d, &chars );
   printf( "You entered %d characters "
           "and %d items.\n", chars, items);
}
```

This is sample output:

```
Enter four integers: 23 408 1038 3
You entered 13 characters and 4 items.
```

Using scan sets

Scan sets let you determine exactly which characters are read into a string argument. In a scan set like `%[...]`, you list just the characters you want to read. In this example, the scan set `%[1234567890]` reads only digits:

```
char s[80];
printf( "Enter a line: " );
scanf( "%[1234567890]", s );
printf( "s = %s\n", s );
```

This is sample output:

```
Enter a line: 274abc348
s = 274
```

In a set like `%[X-X]`, you can give a range of characters you want to read. The range is determined using the ASCII ordering sequence. In this example, the scan set `%[A-Za-z]` reads only letters:

```
char s[80];
printf( "Enter a line: " );
scanf( "%[A-Za-z]", s );
printf( "s = %s\n", s );
```

This is sample output:

```
Enter a line: Tom&Jerry
s = Tom
```


And in a set like `%[^...]`, you can specify the characters you do *not* want to read. In this example, the scan set `%[^0-9]` reads in anything, except digits:

```
char s[80];
printf( "Enter a line: " );
scanf( "%[^0-9]", s );
printf( "s = %s\n", s );
```

This is sample output:

```
Enter a line: abcdef123ghi
s = abcdef
```

Here are some more example scan sets:

This scan set...	Matches the longest string of characters in which...
`[xyz]`	all are x, y, or z.
`%[^xyz]`	none are x, y, or z.
`%[a-z]`	all are lowercase letters (from a to z).
`%[^a-z]`	none are lowercase letters (from a to z).
`%[]xyz]`	all are x, y, z, or the right bracket. (The right bracket must be the first character in the scan set.)
`%[^]xyz]`	none are x, y, z, or the right bracket.
`%[-xyz]`	all are x, y, z, or the hyphen. (The hyphen must be the first or last character in the scan set.)
`%[xyz-]`	all are x, y, z, or the hyphen.
`%[^-xyz]`	none are x, y, z, or the hyphen.

printf Summary

This section summarizes all the options for the different components of a `printf()` specifier. For more information on how to print a specific type of data, see the appropriate section.

Flag character

A flag modifies the specifier in some way. You can use any number of these flags:

Directive	Meaning
- (hyphen)	Left-justify what's printed within the field.
+ (plus sign)	Print a positive number (integer or floating-point) with a plus sign. (Otherwise, `printf()` prints a minus sign only if the number is negative.)
' ' (space)	• Prefix a number (integer or floating-point) with a space if it's positive. • Print a space if the specifier is `d`, `i`, `o`, `u`, `x`, or `X`, the precision is zero, and the value is zero. If you use both the space and + flag, `printf()` ignores the space flag.
# (pound)	Convert the printed value to an alternate form: • For the `o` specifier, prefix the number with 0, by increasing the precision to force the first digit to be zero. • For the `x` (or `X`) specifier, prefix the number with `0x` (or `0X`), if the number isn't zero. • For the `e`, `E`, `f`, `g`, or `G` specifiers, always include a decimal point in the result, even if no digits follow the decimal point. • For the `g` and `G` specifiers, don't remove trailing zeros from the result. • For the `s` specifier, use a Pascal-style string (an object of type `Str255`) as the argument. This is a THINK C extension.

Directive	Meaning
0 (zero)	For the `d`, `i`, `o`, `u`, `x`, `X`, `E`, `f`, `g`, and `G` specifiers, pad the field with leading zeros (inserting them after any indication of sign or base). If there's a precision specification, the `d`, `i`, `o`, `u`, `x`, and `X` specifiers ignore the `0` flag. If the field size is negative, `printf()` ignores the `0` flag.

Field size

The field size specifies the number of characters that the argument fills. It can be one of these:

Directive	Meaning
* (asterisk)	Use the next argument in the function's argument list as the field size. The argument must be an integer.
an integer	Use the integer as the field size.

If the argument has fewer characters than the field size, `printf()` pads the output with spaces. If the field size is positive, it adds the spaces to the right. If the field size is negative, it adds the spaces to the left.

Precision directive

The precision specifies one of these, depending on the conversion character:

If the conversion character is...	The precision directive specifies...
`s`	The maximum number of characters to print from the string.
`e`, `E`, `f`	The number of digits to print after the decimal point.
`g`, `G`	The maximum number of significant digits to print.
`d`, `i`, `o`, `u`, `x`, `X`	The minimum number of digits to print. Remaining space is filled with zeroes.
`c`, `n`, `p`, `%`	The precision has no effect.

The precision is a period (.) optionally followed by one of these:

Directive	Meaning
* (asterisk)	Use the next argument in the function's argument list as the field size. The argument must be an integer.
an integer	Use the integer as the field size.

If there's no precision, the specifiers d, i, o, u, x, and X use a precision of 1, and the specifiers e, E, f, g, and G use a precision of 6. If there's a period, but no integer or asterisk, all specifiers use a precision of 0. A negative precision is ignored.

Note

When you use an asterisk for the field size or precision, make sure the arguments for the specifier are in this order: field size, precision, and the argument to print.

Argument size

The argument size specifies the size of the argument. It can be one of the following:

Directive	Meaning
h	The conversion specifier prints an argument of the following type: • For d or i, `short int`. • For o, u, x, or X, `unsigned short int`. • For e, E, f, g, or G, `short double`.
l (letter ell)	The conversion specifier prints an argument of the following type: • For d or i, `long int`. • For o, u, x, or X, `unsigned long int`.

Directive	Meaning
L	The conversion specifier prints an argument of the following type: • For e, E, f, g, or G, `long double`.

h, l, and L are not valid with any other conversion specifiers.

Conversion characters

The conversion character specifies how to print the argument. These are the conversion characters:

Char	Argument Type	Input
c	`char` `int`	A single character
d	`char` `int`	A signed decimal integer. [-] *dddd*
e, E	`float` `double`	A signed decimal floating point number. [-] *d*.*ddd*e±*xx* (for e) [-] *d*.*ddd*E±*xx* (for E) • This prints exactly one digit before the decimal point. • If the precision is 0, this does not print a decimal point.
f	`float` `double`	A signed decimal fixed-point number. [-] *mmm*.*ddd* • If the precision is 0, this does not print a decimal point. • If this prints a decimal point, it prints at least one digit before the decimal point.

Char	Argument Type	Input
g, G	`float` `double`	Chooses between fixed-point and exponential notation. Uses exponential notation if the exponent is less than -4 or greater than the precision. • Doesn't print trailing zeroes. • Prints a decimal point only if it's followed by numbers.
i	`char` `int`	A signed decimal integer. [-] *dddd* If the value is zero and the precision is zero, this prints nothing.
n	`int *`	Nothing printed. This puts into an integer the number of characters the function has printed so far during this call.
o	`unsigned char` `unsigned int`	An unsigned octal integer. dddd If the value is zero and the precision is zero, this prints nothing.
p	`void *`	An eight-digit hexadecimal value

Char	Argument Type	Input
s	`char *` `Str255`	A string. With the # flag, this specifier prints a Pascal-style string (an object of type `Str255`). Without the # flag, this specifier prints a C-style string (an object of type `char *`). It prints the string until one of the following happens: • It encounters a NULL character, which it doesn't print. • It prints the maximum number of characters allowed by the precision.
u	`unsigned char` `unsigned int`	An unsigned decimal integer.
x, X	`unsigned char` `unsigned int`	An unsigned hexadecimal integer. • If the directive is x, this uses a, b, c, d, e, and f as the extra digits. • If the directive is X, this uses A, B, C, D, E, and F as the extra digits.
%	No argument used	A percent sign (%)

scanf Summary

This section summarizes all the legal values for the different components of a `scanf()` specifier. For more information on how to read a specific type of data, see the appropriate section.

Flag character

The flag character controls whether to ignore the input field that matches the specifier.

Directive	Meaning
* (asterisk)	Don't assign the input field that matches this specifier to an argument.

Field size

The field size directive specifies the width of the input field.

Directive	Meaning
a decimal integer	The width of the field that the specifier reads data from.

Argument size

The argument size directive specifies the size of the argument. These are the argument size directives:

Directive	Meaning
h	The specifier reads in an argument of the following type: • For d, i, or n, `short int *`. • For o, u, or x, `unsigned short int *`. • For e, E, f, g, or G, `short double`.
l (letter ell)	The specifier reads in an argument of the following type: • For d, i, or n, `long int *`. • For o, u, or x, `unsigned long int *`. • For e, f, or g, `double *`.
L	The specifier reads in an argument of the following type: • For e, f, or g, `long double`.

h, l, and L are not valid with any other conversion specifiers.

Conversion characters

The conversion character specifies how to interpret the input field. These are the conversion characters:

Char	Argument Type	Input
c	`integer *` `char *`	A character • Accepts all kinds of characters: white space, control characters, etc. • The argument must point to an object large enough to hold all the characters specified. • Does not add a terminating NULL.
d	`int *`	A decimal integer
e, E, f, g, G	`float *`	A floating-point number with optional sign, decimal point, and exponent integer.
i	`int *`	An integer Figures the base of the number according to its prefix: • 0 means it's octal. • 0x or 0X means it's hexadecimal. • Otherwise, it's decimal.
n	`int *`	No argument read. Puts into its argument the number of characters the function has read so far during this call.
o	`unsigned int *`	An octal integer.
p	`void *`	A hexadecimal number.

Char	Argument Type	Input
s	`char *`	A character string. • Reads only characters that aren't whitespace. • The argument must point to an object that's large enough to hold all the characters. • Adds a terminating null character (`'\0'`).
u	`unsigned int *`	A decimal integer.
x	`unsigned int *`	A hexadecimal integer.
%	No argument filled	A percent sign.
[...]	`char *`	See section "Using scan sets" on page 82.

THINK C

The Standard Functions

Part Three

5 The Standard Library Functions

The Standard Library Functions 5

This chapter describes THINK C's ANSI library and includes a complete catalogue of the functions it contains.

Contents

Using the Function Descriptions . . . 99
Reading the descriptions . . . 99
Using functions not in the ANSI standard . . . 99
abort() . . . 99
abs() . . . 101
acos() . . . 102
asctime() . . . 103
asin() . . . 104
assert() . . . 105
atan() . . . 106
atan2() . . . 107
atexit() . . . 108
_atexit() . . . 109
atof() . . . 111
atoi() . . . 113
atol() . . . 114
bsearch() . . . 115
calloc() . . . 116
ccleol() . . . 119
ccleos() . . . 120
ccommand() . . . 121
cecho2file() . . . 122
cecho2printer() . . . 125
ceil() . . . 126
cgetxy() . . . 128
cgotoxy() . . . 129
chide() . . . 130
cinverse() . . . 131
clearerr() . . . 133
clock() . . . 134
close() . . . 135
cos() . . . 136
cosh() . . . 137
creat() . . . 138
csetmode() . . . 139
. . . 140

csettabs() . . . 142
cshow() . . . 143
ctime() . . . 145
difftime() . . . 146
div() . . . 147
exec(), execl(), execle(), execlp(), execlpe(), execv(), execve(), execvp(), execvpe() . . . 148
exit() . . . 149
_exit() . . . 150
_exiting() . . . 152
exp() . . . 154
fabs() . . . 155
fclose() . . . 156
fdopen() . . . 157
feof() . . . 158
ferror() . . . 159
fflush() . . . 160
fgetc() . . . 162
fgetpos() . . . 164
fgets() . . . 166
fileno() . . . 168
floor() . . . 169
fmod() . . . 170
fopen() . . . 171
fopenc() . . . 174
fprintf() . . . 175
fputc() . . . 177
fputs() . . . 178
fread() . . . 179
free() . . . 181
freopen() . . . 182
freopenc() . . . 184
frexp() . . . 185
fscanf() . . . 187
fseek() . . . 189
fsetpos() . . . 191
ftell() . . . 193
fwrite() . . . 195
getc() . . . 197
getchar() . . . 199
getegid(), geteuid(), getgid(), getpgrp(), getpid(), getppid(), getuid() . . . 201
getenv() . . . 202
getlogin() . . . 203
gets() . . . 204
getw() . . . 205
gmtime() . . . 206
isatty() . . . 207
isxxxx() . . . 208
kill() . . . 210
labs() . . . 211
ldexp() . . . 212

ldiv() . . . 213
localeconv() . . . 214
localtime() . . . 215
log() . . . 216
log10() . . . 217
lseek() . . . 218
longjmp() . . . 220
malloc() . . . 222
mblen() . . . 223
mbstowcs() . . . 224
mbtowc() . . . 225
memchr() . . . 226
memcmp() . . . 227
memcpy() . . . 228
memmove() . . . 229
memset() . . . 230
mktime() . . . 231
modf() . . . 233
open() . . . 234
perror() . . . 236
pow() . . . 237
printf() . . . 238
putc() . . . 239
putchar() . . . 241
puts() . . . 242
putw() . . . 243
qsort() . . . 244
raise() . . . 246
rand() . . . 248
read() . . . 249
realloc() . . . 251
remove() . . . 253
rename() . . . 254
rewind() . . . 255
scanf() . . . 256
setbuf() . . . 258
setjmp() . . . 260
setlocale() . . . 262
setvbuf() . . . 263
signal() . . . 265
sin() . . . 268
sinh() . . . 269
sleep() . . . 270
sprintf() . . . 271
sqrt() . . . 272
srand() . . . 273
sscanf() . . . 274
strcat() . . . 275
strchr() . . . 276
strcmp() . . . 277
strcoll() . . . 278
strcpy() . . . 279

strcspn() 280
strerror() 281
strftime() 282
strlen() 284
strncat() 285
strncmp() 287
strncpy() 289
strpbrk() 290
strrchr() 291
strspn() 292
strstr() 293
strtod() 294
strtok() 296
strtol() 298
strtoul() 300
strxfrm() 302
system() 303
tan() 304
tanh() 305
tell() 306
time() 308
tmpfile() 309
tmpnam() 310
tolower() 311
toupper() 313
ungetc() 315
unlink() 317
va_start(), va_arg(), va_end() 318
vfprintf() 321
vprintf() 323
vsprintf() 325
wctomb() 327
wcstombs() 328
write() 329

Using the Function Descriptions

This chapter describes all the functions in the THINK C standard libraries: ANSI and unix. Before using a function from this chapter in your program, follow these steps:

1. Add the library in section "Library" to your project. If more than one library is listed, see "What Are the Standard Libraries?" on page 11 to decide which one you should include.
2. #Include the header file from the section "Syntax" in your program.

Reading the descriptions

Most function descriptions have these six sections: (Some may omit the sections "Returns," "See Also," and "Example.")

This section...	Includes...
Library	The standard libraries that contain this function. This list can contain any of these libraries: `ANSI`, `ANSI-small`, `ANSI-A4`, or `unix`.
Syntax	Two C statments: the `#include` statement which a file must contain if it uses the function, and the function's prototype which describes the function's return type and arguments.
Description	A description of how to use the function.
Returns	The value that the function return. If the function sets `errno` when there's an error, this section also describes `errno`'s possible values.
See Also	Other related functions. Look at them if this function doesn't do exactly what you want.
Example	A program that illustrates how you might use the function.

Using functions not in the ANSI standard

This chapter marks functions that aren't in the ANSI standard. This symbol appears in the upper-right-hand corner of the page:

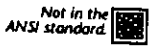

Figure 5-1 The "Not in the ANSI standard" marker

If you plan to compile your program with another ANSI C compiler, avoid the functions that aren't in the standard.

Note that many functions in THINK C's ANSI library aren't in the ANSI standard library. These are functions that are closely related to ANSI standard functions (like _atexit() or _exit()) or work with ANSI standard functions (like the console functions). They provide additional functionality that wasn't included in the standard.

abort

Library

ANSI, ANSI-small

Syntax

```
#include <stdlib.h>
void abort(void);
```

Description

To find out how to handle a signal, see signal() *on page 265.*

abort() terminates your program abnormally. It does the following:

- raises the signal SIGABRT
- executes all functions registered with _atexit()
- flushes all open output streams
- closes all open files
- removes all temporary files.

If your program handles SIGABRT, and the signal handler doesn't return (for example, it calls longjmp()), your program won't terminate.

abort() is similar to _exit(), except that _exit() doesn't raise SIGABRT.

See Also

_atexit(), _exit(), _exiting(), atexit(), exit()

Example

This program exits your program if the error number for HandleError() is 0:

```
#include <stdio.h>
#include <stdlib.h>

main()
{
   HandleError(0);
}

int HandleError( int err )
{
   char s[5];

   if ( err != 0 )
      printf( "Error #%d happened.\n", err );
   else {
      printf( "Fatal error #%d. "
              "Program aborted.\n", err );
      printf( "Press Return to continue.\n" );
      gets( s );
      abort();
   }
}
```

abs

Library

ANSI, ANSI-small, ANSI-A4

Syntax

```
#include <stdlib.h>
int abs(int i);
```

Description

abs() computes the absolute value of its argument.

Returns

| i | for any integer i.

See Also

fabs(), labs()

Example

This program prints the absolute value of any integer you enter:

```
#include <stdio.h>
#include <stdlib.h>

main()
{
   int x, result;

   printf( "Enter a number: " );
   scanf( "%d", &x );

   result = abs(x);

   printf( "|%d| = %d\n", x, result);
}
```

This is sample output from the program:

```
Enter a number: -234
|-234| = 234
```


acos

Library

ANSI

Syntax

```
#include <math.h>
double acos(double x);
```

Description

acos() returns the principal value of the arc cosine ($\cos^{-1}$) of x, for any x in the range −1 to +1.

Returns

acos() returns these values:

If x is...	The function returns...
between −1 and +1	$\cos^{-1}x$ in the range 0 to π radians.
not between −1 and +1	0. It also sets errno to EDOM.

See Also

asin(), atan(), atan2(), cos(), cosh(), sin(), tan()

Example

This program prints the arc cosine of any number you enter:

```
#include <stdio.h>
#include <math.h>

main()
{
   double x, result;

   printf( "Enter a number: " );
   scanf( "%lf", &x );

   result = acos(x);

   printf( "acos(%G) = %G\n", x, result);
}
```

This is sample output from the program:

```
Enter a number: 0.25
acos(0.25) = 1.31812
```

asctime

Library ANSI

Syntax

```
#include <time.h>
char *asctime(const struct tm *timeptr)
```

Description `asctime()` converts into a string the time in `timeptr`. The string has this form (for example):

```
Tue Sep 25 15:23:24 1990\n\0
```

Broken-down time is a structure representing the date and time. It includes these fields: (They're all of type `int`.)

Field	Description (range)
`tm_sec`	Seconds after the minute (0 to 61)
`tm_min`	Minutes after the hour (0 to 59)
`tm_hour`	Hours since midnight (0 to 23)
`tm_mday`	Day of the month (1 to 31)
`tm_mon`	Months since January (0 to 11)
`tm_year`	Years since 1900
`tm_wday`	Days since Sunday (0 to 6)
`tm_yday`	Days since January 1 (0 to 365)
`tm_isdst`	Daylight Savings Time flag

Returns The string containing the time.

See Also `ctime()`, `localtime()`, `mktime()`, `strftime()`, `time()`

Example This program prints the current date and time:

```
#include <stdio.h>
#include <time.h>

main()
{
   time_t now;
   struct tm *date;
   char *s;

   now = time( NULL );
   date = localtime( &now );
   s = asctime( date );

   printf( "%s", s );
}
```


asin

Library ANSI

Syntax

```
#include <math.h>
double asin(double x);
```

Description `asin()` returns the principal value of the arc sine ($\sin^{-1}$) of `x`, for any `x` in the range `-1` to `+1`.

Returns `asin()` returns these values:

If x is...	The function returns...
between -1 and +1	$\sin^{-1}x$ in the range $-\pi/2$ to $+\pi/2$ radians.
not between -1 and +1	0 (zero). It also sets `errno` to `EDOM`

See Also `acos()`, `atan()`, `atan2()`, `cos()`, `sin()`, `sinh()`, `tan()`

Example This program prints the arc sine of any number you enter:

```
#include <stdio.h>
#include <math.h>

main()
{
   double x, result;

   printf( "Enter a number: " );
   scanf( "%lf", &x );

   result = asin(x);

   printf( "asin(%G) = %G\n", x, result);
}
```

This is sample output from the program:

```
Enter a number: .25
asin(0.25) = 0.25268
```

assert

Library

ANSI, ANSI-small

Syntax

```
#include <assert.h>
void assert(int expression);
```

Description

`assert()` is a macro that checks an expression. If it's false (that is, equals 0 (zero)), `assert()` prints an error message to `stderr` and calls `abort()`.

`assert()` lets you put diagnostics in your programs. It reports its failure in the following format:

`Assertion failed:` *expression*, `file` *name*, `line` *num*

`assert()` gets the file name and line number from the macros `__FILE__` and `__LINE__`.

If you want the compiler to ignore your `assert()` calls, put a `#define NDEBUG` statement just before the `#include <assert.h>` statement.

See Also

`abort()`, `perror()`, `strerror()`

Example

This program prints the square root of any valid number than you enter. It uses `assert()` to make sure that your number is not negative.

```
#include <stdio.h>
#include <math.h>
#include <assert.h>

main()
{
   double x, result;

   printf( "Enter a number: " );
   scanf( "%lf", &x );

   assert( x>=0 );

   printf( "√%G = %G\n", x, sqrt(x));
}
```

This is sample output from the program:

```
Enter a number: -2
Assertion failed: x>=0, file main.c, line 12
```


atan

Library

ANSI

Syntax

```
#include <math.h>
double atan(double x);
```

Description

`atan()` returns the principal value of the arc tangent ($\tan^{-1}$) of x.

Returns

$\tan^{-1}x$, in the range $-\pi/2$ to $+\pi/2$ radians.

See Also

`acos()`, `asin()`, `atan2()`, `cos()`, `sin()`, `tan()`, `tanh()`

Example

This program prints the arc tangent of any number you enter:

```
#include <stdio.h>
#include <math.h>

main()
{
   double x, result;

   printf( "Enter a number: " );
   scanf( "%lf", &x );

   result = atan(x);

   printf( "atan(%G) = %G\n", x, result);
}
```

This is sample output from the program:

```
Enter a number: 0.25
atan(0.25) = 0.244979
```

atan2

Library ANSI

Syntax

```
#include <math.h>
double atan2(double y, double x);
```

Description `atan2()` computes the principal value of the arc tangent ($\tan^{-1}$) of `y/x`. It uses the signs of both arguments to determine the quadrant of the return value.

Returns `atan2()` returns these values:

If x and y are...	The function returns
not both 0 (zero)	$\tan^{-1}$ (y/x) in the range $-\pi$ to $+\pi$ radians.
both 0 (zero)	0 (zero). It also sets `errno` to `EDOM`.

See Also `acos()`, `asin()`, `atan()`, `cos()`, `sin()`, `tan()`, `tanh()`

Example This program prints the arc tangent of `x/y` for any two numbers you enter:

```
#include <stdio.h>
#include <math.h>

main()
{
   double x, y, result;

   printf( "Enter x: " );
   scanf( "%lf", &x );

   printf( "Enter y: " );
   scanf( "%lf", &y );

   result = atan2(x,y);

   printf( "atan2( %G/%G ) = %G\n",
           x, y, result);
}
```

This is sample output from the program:

```
Enter x: 3
Enter y: 4
atan2( 3/4 ) = 0.643501
```


atexit

Library ANSI, ANSI-small

Syntax

```
#include<stdlib.h>
int atexit (void (*func) (void));
```

Description `atexit()` lets you execute functions just before your program exits normally. It adds the function `func` to a list of functions that are called without arguments when your program exits normally. You can register at most 32 functions with either `atexit()` or `_atexit()`. The functions are called in reverse of the order you registered them.

`_atexit()` is similar to `atexit()`, except that the functions registered with `_atexit()` execute no matter how your program exits. The functions registered with `atexit()` execute only when your program exits normally. A program exits normally when it returns from `main()` or calls `exit()`. A program exits abnormally when it calls `abort()`, `_exit()`, or `ExitToShell()`.

For more information on `ExitToShell()`, see Inside Macintosh II, Chapter 2, "Segment Loader."

Returns `atexit()` returns these values:

If the function...	It returns...
added the function to the list	Zero
couldn't add the function	Non-zero

See Also `_atexit()`, `_exit()`, `_exiting()`, `abort()`, `exit()`

Example

This program adds two functions to the list of functions that are called when the program exits:

```
#include <stdio.h>
#include <stdlib.h>

void exit1( void )
{
   printf( "This is exit1().\n" );
}

void exit2( void )
{
   printf( "This is exit2().\n" );
}

main()
{
   atexit( exit1 );
   atexit( exit2 );
   printf( "This is main.\n" );
}
```

The program prints these lines:

```
This is main.
This is exit2().
This is exit1().
```


_atexit

Not in the ANSI standard

Library

ANSI, ANSI-small

Syntax

```
#include <stdlib.h>
int _atexit (void (*func) (void));
```

Description

`_atexit()` lets you specify a set of functions that execute just before your program exits whether it ends normally or is aborted. It adds the function `func` to a list of functions that are called without arguments when your program exits. You can register at most 32 functions with either `atexit()` or `_atexit()`. The functions are called in reverse of the order you registered them.

`_atexit()` is similar to `atexit()`, except that the functions registered with `atexit()` execute only when your program exits normally. The functions registered with `_atexit()` execute no matter how your program exits. A program exits normally when it returns from `main()` or calls `exit()`. A program exits abnormally when it calls `abort()`, `_exit()`, or `ExitToShell()`.

For more information on `ExitToShell()`, see Inside Macintosh II, Chapter 2, "Segment Loader."

This function is a THINK C extension that is not in the ANSI standard.

Returns

`_atexit()` returns these values:

If the function...	It returns...
added the function to the list	Zero
couldn't add the function	Non-zero

See Also

`_exit()`, `_exiting()`, `exit()`, `abort()`, `atexit()`

Example

This program adds two functions to the list of functions that are called when the program exits:

```
#include <stdio.h>
#include <stdlib.h>

void exit1(void)
{
   printf( "This is exit1().\n" );
}

void exit2(void)
{
   printf( "This is exit2().\n" );
}

main()
{
   _atexit( exit1 );
   _atexit( exit2 );
   printf( "This is main.\n" );
}
```

The program prints these lines:

```
This is main.
This is exit2().
This is exit1().
```


atof

Library

ANSI

Syntax

```
#include <stdlib.h>
double atof(const char *nptr);
```

Description

`atof()` converts the string `nptr` to a double-precision floating point value. It skips over any whitespace characters at the beginning of `nptr`. It stops converting when it reaches a character that can't be part of a number

`atof()` looks for a number formatted like this:

`[+|-] [digits] [.digits] [d|D|e|E [-|+] digits]`

Anything in brackets (`[]`) is optional. If items are separated by vertical bars (`|`), only one of those items may be in that position.

`atof(nptr)` is equivalent to

```
strtod( nptr, NULL );
```

Returns

The converted double precision floating point value.

See Also

`atoi()`, `atol()`, `strtod()`, `strtol()`, `strtoul()`

Example

This program converts some strings to floating point numbers:

```
#include <stdio.h>
#include <stdlib.h>

main()
{
   char *s1 = "3.141593";
   char *s2 = "123.45E+676HIKE!";
   double result;

   result = atof( s1 );
   printf( "s1 is %G.\n", result );

   result = atof( s2 );
   printf( "s2 is %G.\n", result );
}
```

The program prints this:

```
s1 is 3.141593.
s2 is 1.2345E+678.
```

atoi

Library

ANSI, ANSI-small, ANSI-A4

Syntax

```
#include <stdlib.h>
int atoi(const char *nptr)
```

Description

`atoi()` converts a string to an integer value. It converts the number represented in `nptr` to an integer, skipping over any whitespace characters in the beginning of the string. It stops converting when it reaches a character that can't be part of a number.

`atoi()` looks for a number formatted like this:

`[+|-] [digits]`

Anything in brackets (`[]`) is optional. If items are separated by vertical bars (`|`), only one of those items may be in that position.

`atoi(nptr)` is equivalent this statement, except that `atoi()` handles errors differently:

```
(int) strtol( nptr, (char **)NULL, 10 );
```

Returns

The converted integer.

See Also

`atof()`, `atol()`, `strtod()`, `strtol()`, `strtoul()`

Example

This example converts two strings to integer values:

```
#include <stdio.h>
#include <stdlib.h>

main()
{
   char *s1 = "9823boo!";
   int result;

   result = atoi( s1 );
   printf( "s1 is %d.\n", result );
}
```

The program prints this:

```
s1 is 9823.
```


atol

Library

ANSI, ANSI-small, ANSI-A4

Syntax

```
#include <stdlib.h>
long int atol(const char *nptr);
```

Description

`atol()` converts a string to an long integer value. It converts the number represented in `nptr` to a long integer, skipping over any whitespace characters in the beginning of the string. It stops converting when it reaches a character that can't be part of a number.

`atol()` looks for a number formatted like this:

`[+|-] [digits]`

Anything in brackets (`[]`) is optional. If items are separated by vertical bars (`|`), only one of those items may be in that position.

`atol(nptr)` is equivalent to

```
strtol( nptr, (char **)NULL, 10 );
```

Returns

The converted long integer.

See Also

`atof()`, `atoi()`, `strtod()`, `strtol()`, `strtoul()`

Example

This example converts one string to long integer values:

```
#include <stdio.h>
#include <stdlib.h>

main()
{
   char *s1 = "982397362boo!";
   long int result;

   result = atol( s1 );
   printf( "s1 is %ld.\n", result );
}
```

The program prints this:

```
s1 is 982397362.
```

bsearch

Library

ANSI

Syntax

```
#include <stdlib.h>
void *bsearch(const void *key,
   const void *base,
   size_t nmemb,
   size_t size,
   int (*compar) (const void *key, const void *data));
```

Description

`bsearch()` performs a binary search of a sorted table.

`bsearch()` calls the comparison function `compar()` with two arguments: a pointer to the `key` object and to an array element. Declare the comparison function like this: (You can use different names for the function and arguments.)

```
int compar( const void *key, const void *data );
```

It should return an integer as follows:

If the key object is...	compar returns...
less than the array element	less than zero
equal to the array element	zero
greater than the array element	greater than zero

Note

Make sure that array `base` is sorted in ascending order before passing it to `bsearch()`.

These are the arguments to `bsearch()`:

Argument	Description
`key`	Pointer to the object that `bsearch()` is matching.
`base`	Pointer to the initial element of the array you're searching.
`nmemb`	Number of objects in the array.
`size`	Size of each element in the array.
`compar`	Pointer to the comparison function.

Returns

`bsearch()` returns these values:

If the function...	It returns...
finds a match	a pointer to the element of the array that matches the key
doesn't find a match	NULL

See Also

`qsort()`

Example

This program asks you to enter the name of a dwarf and searches for it in a table of the names for the seven dwarves.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int compare( const void *s1, const void *s2 )
{
   return strcmp( s1, s2 );
}

int FindDwarf( char *name )
{
   char *result;
   static char dwarves[][10] =
     { "Bashful", "Doc", "Dopey", "Grumpy",
       "Happy", "Sleepy", "Sneezy" };

   result = bsearch( name, dwarves, 7,
                     sizeof(char[10]),
                     compare );
   return ( result != NULL );
}

main()
{
   char name[80];
   int found;

   printf( "Enter a name for a dwarf: " );
   scanf( "%s", name );

   if ( FindDwarf(name) )
      printf( "%s is a dwarf.\n", name );
   else
      printf( "%s is not a dwarf.\n", name );
}
```

`strcmp()` is a string comparison function described on page 277. It returns values appropriate for `bsearch()`.

This is output from the program:

```
Enter a name for a dwarf: Dopey
Dopey is a dwarf.
```


calloc

Library

ANSI, ANSI-small

Syntax

```
#include <stdlib.h>
void *calloc(size_t nmemb, size_t size);
```

Description

To learn how THINK C allocates memory, see "Allocating Memory" on page 20.

`calloc()` allocates a block of memory for an array of nmemb objects, each of which is `size` bytes long. It initializes all bits in the block to zero.

To deallocate the block of memory, use `free()`.

Returns

`calloc()` returns these values:

If the function...	It returns...
allocated the block	a pointer to the allocated block of memory
didn't allocate the block	NULL

See Also

`free()`, `malloc()`, `realloc()`

Example

This code fragment allocates a block of memory for an array of structs:

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
   int a, b;
} Foo, *FooPtr;

main()
{
  FooPtr foobuf;
  int size = 100;

  foobuf = (FooPtr) calloc( sizeof(Foo), size );
  if (foobuf == NULL)
    exit();

  foobuf[0].a = 12;
  foobuf[12].b = 100;
  . . .
}
```

ccleol

Not in the ANSI standard.

Library

ANSI

Syntax

```
#include <console.h>
void ccleol(FILE *fp);
```

Description

Clears to the end of the line.

To learn about console windows, see Chapter 3, "Using Console Windows."

`ccleol()` deletes characters from the current cursor position to the end of the line, in the console `fp`.

Example

This example fills a console window with x's, moves the cursor to the middle of a line, and erases the line it's in after you press Return.

```
#include <stdio.h>
#include <console.h>

main()
{
   int i, ch, rows = 12, cols = 40;

   console_options.nrows = rows;
   console_options.ncols = cols;
   cshow( stdout );
   csetmode( C_NOECHO, stdout );
      /* Set mode to C_NOECHO, so        */
      /* getchar() doesn't move cursor. */

   for( i=0; i<rows*cols; i++ )
      putchar( 'x' );
   cgotoxy( 5, 3, stdout );
   ch = getchar();

   ccleol( stdout );
}
```

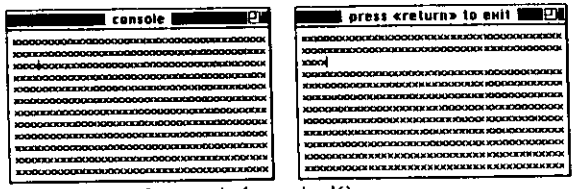

Figure 5-2 Before and after ccleol()

ccleos

Not in the ANSI standard.

Library

ANSI

Syntax

```
#include <console.h>
void ccleos(FILE *fp);
```

Description

Clears to the end of the screen.

To learn about console windows, see Chapter 3, "Using Console Windows."

`ccleos()` erases lines from the line containing the cursor to the end of the screen, in the console `fp`. It leaves the cursor at the start of the first erased line.

Example

This example fills a console window with x's, moves the cursor to the middle of a line, and erases from the line the cursor is in to the end of the window after you press Return.

```
#include <stdio.h>
#include <console.h>

main()
{
   int i, ch, rows = 12, cols = 40;

   console_options.nrows = rows;
   console_options.ncols = cols;
   cshow( stdout );
   csetmode( C_NOECHO, stdout );
      /* Set mode to C_NOECHO, so        */
      /* getchar() doesn't move cursor. */

   for( i=0; i<rows*cols; i++ )
      putchar( 'x' );
   cgotoxy( 5, 3, stdout );
   ch = getchar();

   ccleos( stdout );
}
```

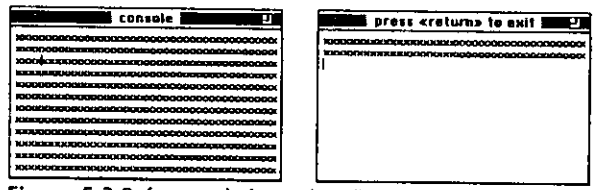

Figure 5-3 Before and after ccleos()

ccommand

Not in the ANSI standard.

Library

ANSI

Syntax

```
#include <console.h>
int ccommand(char ***p);
```

Description

To learn about console windows, see Chapter 3, "Using Console Windows."

Gets a command line.

`ccommand()` displays a dialog that lets you enter command-line arguments and redirect `stdin` (standard input) and `stdout` (standard output). It helps you port programs that require command-line arguments. You'll also find it useful for redirecting input and output for other programs you write.

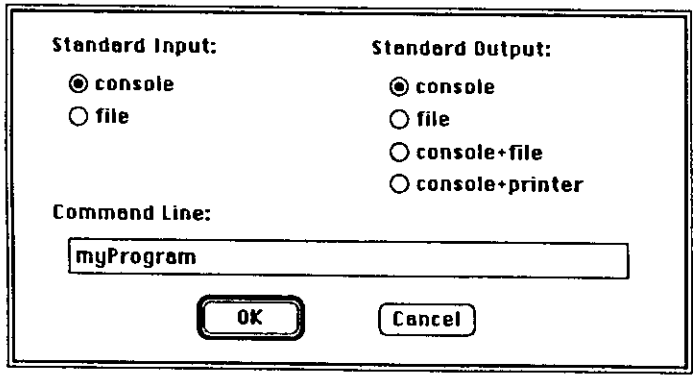

Figure 5-4 The ccommand() dialog.

These are your choices for redirecting standard input:

console	The console package uses a console window as `stdin`.
file	`ccommand()` lets you choose a file from a standard file dialog and redirects `stdin` to it.

These are your choices for redirecting standard output:

console	The console package uses a console window as `stdout`.
file	`ccommand()` lets you create a file from a standard file dialog and redirects `stdout` to it.
console+file	`ccommand()` lets you create a file from a standard file dialog. The console package writes output for `stdout` to both a console window and a file.
console+printer	The console package writes output for `stdout` to a console window. When your program exits, it prints the console's contents to the printer selected in the Chooser.

If your program needs to know whether someone redirected standard input or output to a file, use the function `isatty()` on page 207.

Note

Usually, your application's default folder is the same folder that the application is in. However, if you use `ccommand()` and choose an input or output file that isn't in the same folder as your application, `ccommand()` changes the default folder. You can change the default folder with the function `PBHSetVol()`, described in *Inside Macintosh IV*, Chapter 19, "File Manager."

The Command Line box lets you enter arguments for your program, which are put into the array, `argv`. When you exit the `ccommand()` dialog, `argc` contains the number of command-line arguments, and `argv` is an array of strings with `argc` members. `argv[0]` contains a string that has the name of your program, and the rest of `argv` has strings with your arguments.

Note

You cannot redirect standard input and output in the Command Line box. You must use the radio buttons. For example, entering `MyProgram > MyOutput` into the Command Line box won't redirect output. It will pass > and `MyOutput` as arguments to your program.

Returns The number of command-line arguments, `argc`.

See Also `isatty()`

Example This program prints out the command line you enter in the `ccommand()` dialog:

```
#include <stdio.h>
#include <console.h>

main(int argc, char **argv)
{
   int i;

   argc = ccommand(&argv);

   for (i=0; i<argc; i++)
      printf ("%s ", argv[i]);

   printf ("\n");
}
```


cecho2file

Not in the ANSI standard.

Library `ANSI`

Syntax

```
#include <console.h>
void cecho2file(char *file, int append, FILE *fp);
```

Description Echoes output to a file.

To learn about console windows, see Chapter 3, "Using Console Windows."

After you call `cecho2file()`, the console echoes to the file `file` all complete lines you write to the console `fp`. Once you start echoing, you can't stop it.

To learn how to specify file names, see "Naming files" on page 18.

This table describes what happens to `file`:

If the file...	and append is...	The file is...
exists	non-zero	appended to
exists	zero	overwritten
doesn't exist	zero or non-zero	created

Example This program echoes its output to a file named `log file`.

```
#include <stdio.h>
#include <console.h>

main()
{
   cshow( stdout );
   cecho2file( "log file", 0, stdout );
   printf( "hello world\n" );
}
```

cecho2printer

Not in the ANSI standard.

Library

ANSI

Syntax

```
#include <console.h>
void cecho2printer(FILE *fp);
```

Description

Echoes a console window's output to a printer.

To learn about console windows, see Chapter 3, "Using Console Windows."

After you call `cecho2printer()`, all complete lines you write to the console `fp` are echoed to the printer selected in the Chooser desk accessory.

Example

You can use `cecho2printer()` to print a text file. This program creates a console, opens the file `MyFile`, calls `cecho2printer()`, and prints the file.

```
#include <stdio.h>
#include <stdlib.h>
#include <console.h>

main()
{
   FILE *consolePtr, *filePtr;
   char c;

   /*
    * Open the console
    */
   consolePtr = fopenc();
   if (consolePtr == NULL)
      exit(-1);
   chide (consolePtr);
      /* make the console invisible */
   console_options.pause_atexit = 0;
      /* don't bother to wait */

   /*
    * Open the file
    */
   filePtr = fopen( "MyFile", "r" );
   if (filePtr == NULL)
      exit(-1);
```



```
   /*
    * Print the file
    */
   cecho2printer (consolePtr);
      /* echo to the printer whatever */
      /* you write to the console     */

      /* print the file a character at a time */
   while ( (c=fgetc(filePtr)) != EOF )
      putc( c, consolePtr );

   /*
    * Close the console and file.
    */
   fclose( consolePtr );
   fclose( filePtr );
}
```

ceil

Library ANSI

Syntax

```
#include <math.h>
double ceil(double x);
```

Description `ceil()` rounds x up to the next highest integer. For example, `ceil(1.1)` returns `2.0`, `ceil(4.9)` returns `5.0`, and `ceil(-2.1)` returns `-2.0`.

Returns A double that is the next highest integer from x.

See Also `floor()`, `modf()`

Example This program prints the ceiling of any number you enter:

```
#include <stdio.h>
#include <math.h>

main()
{
   double x, result;

   printf( "Enter a number: " );
   scanf( "%lf", &x );

   result = ceil(x);

   printf( "ceil(%G) = %G\n", x, result);
}
```

This is sample output from the program:

```
Enter a number: 2.7
ceil(2.7) = 3

Enter a number: -5.4
ceil(-5.4) = -5
```


cgetxy

Not in the ANSI standard.

Library ANSI

Syntax

```
#include <console.h>
void cgetxy(int *x, int *y, FILE *fp);
```

Description Reports the cursor position.

To learn about console windows, see Chapter 3, "Using Console Windows."

`cgetxy()` stores the current position of the cursor in console `fp` into x and y, where x holds the column and y holds the row. The upper left corner is at `1, 1`. The lower right corner is `console_options.ncols, console_options.nrows`, which is `80, 25` by default.

See Also `cgotoxy()`

Example This example greets you, saves the position of the cursor, writes that position, and moves back to that position:

To learn about `console_options`, read "Setting console options" on page 44.

```
#include <stdio.h>
#include <console.h>

main()
{
   int x, y;

   console_options.nrows = 12;
   console_options.ncols = 40;
   cshow( stdout );
   cgotoxy( 1, 1, stdout );

   printf( "Hello." );

   cgetxy( &x, &y, stdout );

   printf( "\n x = %d, y = %d\n", x, y );
   cgotoxy( x, y, stdout );
}
```

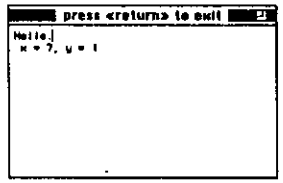

Figure 5-5 After moving the cursor to the saved position.

cgotoxy

Not in the ANSI standard.

Library

ANSI

Syntax

```
#include <console.h>
void cgotoxy(int x, int y, FILE *fp)
```

Description

To learn about console windows, see Chapter 3, "Using Console Windows."

For more information on `console_options`, *read "Setting console options" on page 44.*

Sets the cursor position.

`cgotoxy()` positions the cursor in the console `fp` at `x` and `y`, where `x` is the column and `y` is the row. The upper left corner is `1, 1`. The lower right corner is `console_options.ncols, console_options.nrows`, which is `80, 25` by default.

Warning

`cgotoxy()` does not check its arguments. Be careful not to put the cursor off the screen.

See Also

`cgetxy()`

Example

This code fragment moves the cursor to the center of the standard output console and writes "Hello."

```
#include <stdio.h>
#include <console.h>

main()
{
   int rows = 12, cols = 40;

   console_options.nrows = rows;
   console_options.ncols = cols;
   cshow( stdout );

   cgotoxy(cols/2, rows/2, stdout);
   printf( "Hello" );
}
```

Figure 5-6 "Hello" in the middle of the screen.

chide

Not in the ANSI standard.

Library

ANSI

Syntax

```
#include <console.h>
void chide(FILE *fp);
```

Description

To learn about console windows, see Chapter 3, "Using Console Windows."

Hides a console.

`chide()` hides the console `fp`. You can show it again with `cshow()`.

Note

`chide()` does not close the console. When you show the console again, it retains its settings and text. To close a console, use `fclose()`, described in Chapter 3, "ANSI Standard Functions."

If you write to a hidden console window , the console package won't show it, but will display what you've written when the window reappears. If you try to read from a hidden console window, the console package shows it and brings it to the front.

See Also

`cshow()`

Example

This program creates two windows, hides one, and then shows it again. Notice that the program writes a message in the console window before the program hides it. When the window reappears, the message is still there. Also notice that you have to press Return before the program hides the window, before it shows the window, and before it ends.

```
#include <console.h>
#include <stdio.h>

main()
{
   FILE *c1, *c2;

   console_options.pause_atexit = 0;
   console_options.nrows = 6;
   console_options.ncols = 20;
   console_options.title = "\pconsole 1";
   c1 = fopenc();

   console_options.top = 100;
   console_options.left = 20;
   console_options.title = "\pconsole 2";
   c2 = fopenc();
```

```
    cgotoxy( 1, 1, c1 );
    fprintf( c1, "Hello World.\n" );
    getc( c2 );

    chide( c1 );

    getc( c2 );
    cshow( c1 );
    getc( c2 );
}
```

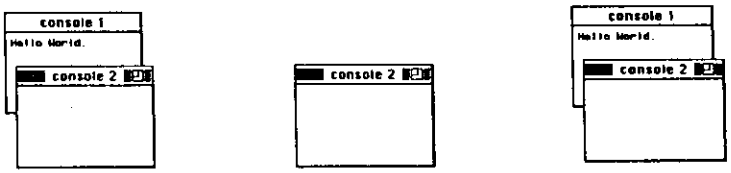

Figure 5-7 Pictures of the screen, after writing a message to console 1, after hiding it, and after showing it again

cinverse

Not in the ANSI standard.

Library

ANSI

Syntax

```
#include <console.h>
void cinverse(int x, FILE *fp);
```

Description

To learn about console windows, see Chapter 3, "Using Console Windows."

Enables and disables inverse video mode.

`cinverse()` enables and disables inverse video mode for the console `fp`, according to the value of `x`:

Value	Description
Zero	disables inverse video mode.
Non-zero	enables inverse video mode.

Also, `cinverse()` erases the all the console's text and moves the cursor to the beginning of the line it was in.

To write an inverted character, turn on the inverse video mode and write the character with its high bit set. To set its high bit, OR it with `0x80`.

When you turn on the inverse video mode, keep these notes in mind:

- You cannot write `'Ä'` (or `0x80`) or any character after it in the Macintosh character set, since they have their high bits set. This includes all the international characters, such as *ä*, *é*, *î*, *ö*, and *ü*. For example, the function `putchar ('À')` does not write *À* but instead writes an inverted *K*.
- A non-printing character, such as Tab or Return, must have its high bit clear so the console recognizes and handles it correctly.

Example

This fragment writes what you see in Figure 5-8: *a* and *c* in inverse video, and *b* in normal video:

```
cinverse (TRUE, stdout);
putchar ('a'|0x80);
putchar ('b');
putchar ('c'|0x80);
```


Figure 5-8 *a* and *c* in inverse video, *b* in normal video.

clearerr

Library

ANSI, ANSI-small

Syntax

```
#include <stdio.h>
void clearerr(FILE *stream);
```

Description

`clearerr()` clears the end-of-file and error indicators for the stream `stream`.

See Also

`ferror()`, `feof()`, `perror()`, `rewind()`

Example

This program causes an error in a file, and then clears it.

```
#include <stdio>

void CheckFile( FILE * );

main()
{
   FILE *fp;
   char c;

   fp = fopen( "MyFile", "w+" );
   CheckFile( fp );

   /* This will cause an error. */
   c = fgetc( fp );
   CheckFile( fp );

   /* This will clear it.       */
   clearerr( fp );
   CheckFile( fp );

   fclose( fp );
}

void CheckFile( FILE *fp )
{
   if ( ferror(fp) || feof(fp) )
      printf( "There's an error.\n" );
   else
      printf( "There's no error.\n" );
}
```

The program prints these lines:

```
There's no error.
There's an error.
There's no error.
```


clock

Library

ANSI

Syntax

```
#include <time.h>
clock_t clock(void)
```

Description

`clock()` determines the amount of time since your system started, in clock ticks. It's useful for calculating durations.

To convert clock ticks to seconds, divide by `CLOCKS_PER_SEC`. For example, the amount of time in seconds since your system started is

```
clock() / CLOCKS_PER_SEC
```

Returns

the amount of time since your system started

See Also

`time()`

Example

This program returns the amount of time it takes to square the numbers from 1 to 100,000:

```
#include <stdio.h>
#include <time.h>

main()
{
   clock_t start, secs;
   double i, result;

   start = clock();

   for ( i=1; i<=100000; i++ )
      result = i * i;

   secs = (clock() - start) / CLOCKS_PER_SEC;

   printf( "That took %lu seconds.\n", secs );
}
```

This is sample output from the program:

```
That took 28 seconds.
```

close

Not in the ANSI standard

Library unix

Syntax

```
#include <unix.h>
int close(int fn);
```

Description

To learn how the standard libraries handle streams, see "Using Files and Streams" on page 15.

`close()` closes the file with file number `fn`. The file number is what `open()` or `creat()` returns when it opens the file.

`close()` is similar to `fclose()` on page 156, except that `fclose()` takes a file descriptor for an argument instead of a file number.

Returns

`close()` returns these values:

If the function...	It returns...
succeeds	0
fails	-1

See Also `open()`, `creat()`, `read()`, `write()`.

cos

Library ANSI

Syntax

```
#include <math.h>
double cos(double x);
```

Description `cos()` returns the cosine of the argument `x`.

Returns cos x, in radians.

See Also `acos()`, `asin()`, `atan()`, `atan2()`, `cosh()`, `sin()`, `tan()`

Example This program prints the cosine of any number you enter:

```
#include <stdio.h>
#include <math.h>

main()
{
   double x, result;

   printf( "Enter an angle: " );
   scanf( "%lf", &x );

   result = cos(x);

   printf( "cos(%G) = %G\n", x, result);
}
```

This is sample output from the program:

```
Enter an angle: 1.0
cos(1) = 0.540302
```

cosh

Library

ANSI

Syntax

```
#include <math.h>
double cosh(double x);
```

Description

`cosh()` returns the hyperbolic cosine of the argument x.

Returns

`cosh()` returns these values:

If the function...	It returns...
succeeds	cosh x
fails because cosh x is too large	Infinity. It also sets `errno` to ERANGE.

See Also

`acos()`, `cos()`, `sinh()`, `tanh()`

Example

This program prints the hyperbolic cosine of any number you enter:

```
#include <stdio.h>
#include <math.h>

main()
{
   double x, result;

   printf( "Enter an angle: " );
   scanf( "%lf", &x );

   result = cosh(x);

   printf( "cosh(%G) = %G\n", x, result);
}
```

This is sample output from the program:

```
Enter an angle: 25.0
cosh(25) = 3.60024E+10
```


creat

Not in the ANSI standard

Library

unix

Syntax

```
#include <unix.h>
int creat(char *filename, int mode);
```

Description

To learn how the standard libraries handle streams, see "Using Files and Streams" on page 15.

To learn how to specify file names, see "Naming files" on page 18.

`creat()` creates and opens a new file with name `filename` and returns its file number. If the file already exists, `creat()` overwrites it. `creat()` ignores the mode argument.

To decide whether to create a binary or text file, `creat()` uses the global variable `_fmode`, declared in `fcntl.h`. The initial value of `_fmode` is `O_BINARY`, so `creat()` creates binary files. To let `creat()` create text files, set `_fmode` to `O_TEXT`.

`creat( name, mode )` is identical to

```
open( name, O_WRONLY | O_CREAT | O_TRUNC )
```

Also, `creat( name, mode )` is similar to

```
fopen( name, "wb" ); /* if _fmode is O_BINARY */
```

or

```
fopen( name, "w" );  /* if _fmode is O_TEXT   */
```

except that `fopen()` returns a file descriptor instead of a file number.

Returns

`creat()` returns these values:

If the function...	It returns...
succeeds	the file number of the new file
fails	EOF

See Also

`open()`, `close()`, `read()`, `write()`.

csetmode

Not in the ANSI standard.

Library

ANSI

Syntax

```
#include <console.h>
void csetmode(int mode, FILE *fp);
```

Description

Sets the input mode for a console.

To learn about console windows, see Chapter 3, "Using Console Windows."

These are the arguments to `csetmode()`:

Argument	Description
mode	The input mode for the console. mode can be one of the four values described below.
fp	A console.

The argument mode can be one of these four values:.

For more information on these modes, read "Setting the input mode" on page 45.

Mode	Buffering	Echoing	Editing
C_ECHO	line-buffered	echoed	full
C_NOECHO	line-buffered	none	partial
C_CBREAK	none	echoed	none
C_RAW	none	none	none

When a console is created, the input mode is initially C_ECHO.

Note

`csetmode()` does not affect characters that are already buffered. It affects subsequent input only.

Example

This example sets the mode to C_NOECHO and then asks you to enter a secret word. When you enter your guess, the program doesn't print what you type.

```
#include <stdio.h>
#include <string.h>
#include <console.h>

main()
{
   char guess[80], *secret = "foo";
   int i = 0;

   cshow( stdin );

   csetmode( C_NOECHO, stdin );

   printf( "What's the secret word? " );
   gets( guess );
   if (strcmp( guess, secret ) == 0)
      printf( "\nYes, that's right.\n" );
   else
      printf( "\nSorry, that's wrong.\n" );
}
```

This is sample output:

```
What's the secret word?
Yes, that's right.
```

csettabs

Not in the ANSI standard.

Library

ANSI

Syntax

```
#include <console.h>
void csettabs(int n, FILE *fp);
```

Description

To learn about console windows, see Chapter 3, "Using Console Windows."

Sets tab stops.

`csettabs()` sets tab stops every n spaces in the console `fp`.

When a console is created, tabs stops are initially set every 8 spaces.

Note

`csettabs()` does not change text already displayed. It affects only new output.

Example

This code fragment:

```
csettabs( 9, stdout );
   /* prints a, tab, b, tab, c */
printf( "a\tb\tc" );

csettabs( 4, stdout );
   /* prints a, tab, b, tab, c */
printf( "a\tb\tc" );
```

writes this to standard output:

```
a         b         c
a    b    c
```

Note that there are nine spaces between the letters in the first line and four spaces between them in the second.

cshow

Not in the ANSI standard.

Library

ANSI

Syntax

```
#include <console.h>
void cshow(FILE *fp);
```

Description

To learn about console windows, see Chapter 3, "Using Console Windows."

Shows and selects a console.

`cshow()` shows the console `fp` (if it was hidden) and brings it to the front. If the console is already at the front, `cshow()` does nothing.

`cshow()` is also useful for creating the standard I/O console window before you use the standard streams. For more information, see "Using your windows in the console interface" on page 43.

See Also

`chide()`

Example

This example creates three console windows and then brings the second to the front.

```
#include <console.h>
#include <stdio.h>

main()
{
   FILE *c1, *c2, *c3;

   console_options.pause_atexit = 0;
   console_options.nrows = 6;
   console_options.ncols = 20;
   console_options.title = "\pconsole 1";
   c1 = fopenc();

   console_options.top = 75;
   console_options.left = 20;
   console_options.title = "\pconsole 2";
   c2 = fopenc();
```

```
    console_options.top = 100;
    console_options.left = 30;
    console_options.title = "\pconsole 3";
    c3 = fopenc();

    cshow( c2 );
    getc( c2 );
}
```

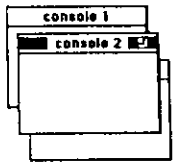

Figure 5-9 console 2 brought to the front.

ctime

Library

ANSI

Syntax

```
#include <time.h>
char *ctime(const time_t *timer);
```

Description

`ctime()` converts to a string the calendar time `timer`. It returns a 26-character string in the form (for example):

```
Tue Sep 25 18:54:25 1990\n\0
```

In THINK C, the calendar time is the number of seconds since Midnight, January 1, 1904.

`ctime(timer)` is equivalent to

```
asctime(localtime(timer))
```

Returns

The string containing the time.

See Also

`asctime()`, `mktime()`, `strftime()`, `time()`

Example

This program prints the current date and time:

```
#include <stdio.h>
#include <time.h>

main()
{
    time_t now;
    char *s;

    now = time( NULL );
    s = ctime( &now );

    printf( "%s", s );
}
```

difftime

Library

ANSI

Syntax

```
#include <time.h>
double difftime(time_t time1, time_t time0);
```

Description

`difftime()` returns the difference between the two calendar times, `time1` and `time0`.

In THINK C, the calendar time is the number of seconds since midnight, January 1, 1904.

Returns

The difference between two calendar times, in seconds, as a double.

See Also

`mktime()`, `time()`

Example

This program prints the number of seconds between now and the year 2000.

```
#include <stdio.h>
#include <time.h>

main()
{
    struct tm date;
    time_t now, then;
    double diff;

    /*
     * Set up date for Midnight, January 1, 2000.
     */
    date.tm_sec = 0;
    date.tm_min = 0;
    date.tm_hour = 0;
    date.tm_mday = 1;
    date.tm_mon = 0;
    date.tm_year = 100;
    date.tm_isdst = -1;

    then = mktime( &date );
    now = time( NULL );
    diff = difftime( then, now );

    printf( "There are %G seconds "
            "until 2000AD.\n", diff );
}
```

For more information on objects of type `struct tm` *see "Telling Time" on page 21.*

This is sample output from the program:

```
There are 2.77219E+08 seconds until 2000AD.
```


div

Library

ANSI, ANSI-small, ANSI-A4

Syntax

```
#include <stdlib.h>
div_t div(int numer, int denom)
```

Description

`div()` computes the quotient and remainder for the division of `numer` by `denom` It returns a structure of type `div_t`, which has these fields:

Field	Type	Description
`quot`	`int`	The quotient
`rem`	`int`	The remainder

This is how `quot` and `rem` are related to `numer` and `denom`:

```
quot * denom + rem == numer.
```

Returns

A structure with the results of the division.

See Also

`fmod()`, `ldiv()`

Example

This program finds the quotient and remainder for `32452 / 787`.

```
#include <stdio.h>
#include <stdlib.h>

main ()
{
    int n = 32452, d = 787;
    div_t r;

    r = div(n,d);

    printf( "%d / %d is %d\n", n, d, r.quot );
    printf( "with a remainder of %d.\n", r.rem);
}
```

The program prints these lines:

```
32452 / 787 is 41
with a remainder of 185.
```

exec
execl
execle
execlp
execlpe
execv
execve
execvp
execvpe

Not in the ANSI standard.

Library

unix

Syntax

```
#include <unix.h>
int exec(char *path, ...);
int execl(char *path);
int execle(char *path);
int execv(char *path, char argv[]);
int execve(char *path, char *argv[], char *envp[]);
int execvp(char *path, ...);
int execvpe(char *path, ...);
```

Description

To learn how to specify file names, see "Naming files" on page 18.

These functions launch the application named in `path`, if they can find it, and exit your program. If they can't find the application, they still exit your program. They ignore all arguments except `path`.

Returns

Nothing. (These functions always exit your program.)

Example

This program launches the application `foo`, if it's in the same folder as this program.

```
#include <unix.h>

main()
{
   exec( "foo" );
}
```


exit

Library

ANSI, ANSI-small

Syntax

```
#include<stdlib.h>
void exit (int status);
```

Description

`exit()` terminates your program normally. It does the following:

- executes all functions registered with `_atexit()` or `atexit()`
- flushes all open output streams
- closes all open files
- removes all temporary files.

The argument `status` is ignored.

See Also

`_atexit()`, `_exit()`, `_exiting()`, `abort()`, `atexit()`

Example

This program exits your program if the error number for `HandleError()` is 0:

```
#include <stdio.h>
#include <stdlib.h>

main()
{
   HandleError(0);
}

int HandleError( int err )
{
   if ( err != 0 )
      printf( "Error #%d happened.\n", err );
   else {
      printf( "Fatal error #%d. "
              "Program aborted.\n", err );
      exit(0);
   }
}
```

_exit

Not in the ANSI standard.

Library

ANSI, ANSI-small

Syntax

```
#include <stdlib.h>
void _exit (int status);
```

Description

`_exit()` terminates your program abnormally. It does the following:

- executes all functions registered with `_atexit()`
- flushes all open output streams
- closes all open files
- removes all temporary files.

The argument `status` is ignored.

`_exit()` is similar to `abort()`, except that `abort()` raises the signal `SIGABRT` before exiting.

This function is a THINK C extension that is not in the ANSI standard.

See Also

`_atexit()`, `_exiting()`, `exit()`, `abort()`, `atexit()`

Example

This program exits your program if the error number for `HandleError()` is 0:

```
#include <stdio.h>
#include <stdlib.h>

main()
{
   HandleError(0);
}

int HandleError( int err )
{
   char s[5];

   if ( err != 0 )
      printf( "Error #%d happened.\n", err );
   else {
      printf( "Fatal error #%d. "
              "Program aborted.\n", err );
      printf( "Press Return to continue.\n" );
      gets( s );
      _exit(0);
   }
}
```


The program prints these lines:

```
Fatal error #0. Program aborted.
Press Return to continue.
```

_exiting

Not in the ANSI standard.

Library

ANSI, ANSI-small

Syntax

```
#include <stdlib.h>
void _exiting(int i);
```

Description

For more information on `_Launch`*, see Inside Macintosh II, Chapter 2, "Segment Loader."*

`_exiting()` prepares a program for exit, but it doesn't actually exit the program. Use `_exiting()` when you use a routine that exits a program without calling the functions registered with `_atexit()` or `atexit()`, like `_Launch`.

`_exiting()` does the following:

- executes the functions registered for either a normal or abnormal exit, depending on the value of `i`
- flushes all open output streams
- closes all open files
- removes all temporary files.

`_exiting()` can prepare your program for either a normal or abnormal exit, depending on the value of `i`:

If i is...	It prepares you for a...
`EXIT_SUCCESS`	Normal exit. It executes the functions registered with `_atexit()` or `atexit()`.
`EXIT_FAILURE`	Abnormal exit. It executes the functions registered with `_atexit()`.

This function is a THINK C extension that is not in the ANSI standard.

See Also

`_atexit()`, `_exit()`, `abort()`, `atexit()`

Example

This program launches a program called `MyApp`, after calling `_exiting()`:

```
#include <stdlib.h>

main()
{
   struct {
      char *appName;
      int memUse;
   } launchInfo = { "\pMyApp", 0 };

   _exiting(1);

   /* There's no glue for _Launch, so you */
   /* must call it from assembly.         */
   asm {
      lea      launchInfo, A0
      _Launch
   }
}
```

exp

Library

ANSI

Syntax

```
#include <math.h>
double exp(double x);
```

Description

exp() returns e^x (e to the x^{th} power). e is the base of natural logarithms.

Returns

exp() returns these values:

If the function...	it returns...
succeeds	e^x
fails because e^x is too large	Infinity. It also sets errno to ERANGE.

See Also

frexp(), ldexp(), log(), log10(), pow(), sqrt()

Example

This program prints e^x for any x you enter:

```
#include <stdio.h>
#include <math.h>

main()
{
   double x, result;

   printf( "Enter a number: " );
   scanf( "%lf", &x );

   result = exp(x);

   printf( "e raised to %G is %G.\n",
           x, result);
}
```

This is sample output from the program:

```
Enter a number: 3.9
e raised to 3.9 is 49.4024.
```


fabs

Library

ANSI

Syntax

```
#include <math.h>
double fabs(double x);
```

Description

fabs() computes the absolute value of a floating-point number, x.

Returns

| x |, for any value x.

See Also

abs(), labs()

Example

This program prints the absolute value of any floating-point number you enter:

```
#include <stdio.h>
#include <math.h>

main()
{
   double x, result;

   printf( "Enter a number: " );
   scanf( "%lf", &x );

   result = fabs(x);

   printf( "fabs(%G) = %G.\n", x, result);
}
```

This is sample output from the program:

```
Enter a number: -3.2e78
fabs(-3.2E+78) = 3.2E+78.
```

fclose

Library

ANSI, ANSI-small

Syntax

```
#include <stdio.h>
int fclose(FILE *stream);
```

Description

To learn how the standard libraries handle streams, see "Using Files and Streams" on page 15.

To learn about console windows, see Chapter 3, "Using Console Windows."

`fclose()` closes the stream `stream`. It closes both files and consoles windows.

Before closing the stream, `fclose()` does the following:

- flushes the stream
- discards unused buffered data
- deallocates the file buffer, if it was allocated automatically. You allocate a buffer automatically when you call `setbuf()` without a buffer or `fopen()`.

Returns

`fclose()` returns these values:

If the function...	It returns...
succeeds	0 (zero)
fails	EOF

See Also

`fopen()`, `close()`, `fopenc()`

Example

This program opens the file `MyFile`, writes to it, and closes it.

```
#include <stdio.h>

main()
{
   FILE *fp;

   fp = fopen( "MyFile", "w" );
   fputs( "Hello file", fp );

   fclose( fp );
}
```


fdopen

Not in the ANSI standard.

Library

unix

Syntax

```
#include <unix.h>
FILE *fdopen(int fn, char *mode);
```

Description

`fdopen()` lets you convert a UNIX file number to an ANSI file descriptor. It returns a pointer to the file descriptor for `fn`, where `fn` is a file number. The argument `mode` is ignored.

Returns

A pointer to the file descriptor for the file with file number `fn`.

See Also

`fileno()`

Example

To learn how the standard libraries handle streams, see "Using Files and Streams" on page 15.

This program opens `MyFile` with `open()`, which returns a file number. It converts the file number to a pointer to a file descriptor, writes a message to the file, and closes it.

```
#include <stdio.h>
#include <fcntl.h>
#include <unix.h>

main()
{
   int fn;
   FILE *fp;
   fn = open( "MyFile", O_WRONLY | O_CREAT |
                        O_TRUNC | O_TEXT );

   fp = fdopen( fn, 0 );

   fprintf( fp, "hello world\n" );
   fclose( fp );
}
```

feof

Library

ANSI, ANSI-small

Syntax

```
#include <stdio.h>
int feof(FILE *stream);
```

Description

To learn how the standard libraries handle streams, see "Using Files and Streams" on page 15.

`feof()` tests whether a previous I/O call to the stream `stream` caused an end-of-file error (such as trying to read a character after you reach the end of the file). It tests the stream's EOF indicator.

If you do any of the following between reaching the end of a file and calling `feof()`, `feof()` will report that you're not at the end of the file:

- Using `clearerr()` to clear the stream's error flag
- Calling `fseek()`.

Returns

`feof()` returns these values:

If you are...	This function returns...
at the end of the file	non-zero
not at the end of the file	zero

See Also

`clearerr()`, `ferror()`, `perror()`, `rewind()`

Example

This program tries to read a character from an empty file and tests whether it reached the end of the file:

```
#include <stdio.h>

main()
{
   FILE *fp;
   char c;

   fp = fopen( "MyFile", "w+" );
   c = fgetc( fp );

   if ( feof(fp) )
     printf( "Reached the end of the file.\n" );
   else
     printf( "You read %c.\n", c );

   fclose( fp );
}
```


ferror

Library

ANSI, ANSI-small

Syntax

```
#include <stdio.h>
int ferror(FILE *stream);
```

Description

To learn how the standard libraries handle streams, see "Using Files and Streams" on page 15.

`ferror()` returns whether an error has occurred on the stream `stream`. It tests the error indicator for the stream.

To clear the error, use `clearerr()`. Calling `rewind()` clears the error indicator for the stream automatically.

`ferror()` does not test for end-of-file errors. To test for those errors, use `feof()`.

Returns

`ferror()` returns these values:

If the stream...	The function returns...
has an error flag set	non-zero
doesn't have an error flag set	zero

See Also

`feof()`, `clearerr()`, `perror()`, `rewind()`

fflush

Library ANSI, ANSI-small

Syntax

```
#include <stdio.h>
int fflush(FILE *stream)
```

Description `fflush()` flushes the I/O buffer of the stream `stream`. It lets you switch cleanly between reading and writing on the same stream.

To learn how the standard libraries handle streams, see "Using Files and Streams" on page 15.

If your program was writing to `stream`, `fflush()` writes out any buffered data to the file. If your program was reading from, `fflush()` discards any buffered data.

If `stream` is NULL, `fflush()` flushes all the streams opened for writing.

Returns `fflush()` returns these values:

If the function...	It returns...
succeeds	0
fails	EOF

See Also `fseek()`, `fsetpos()`, `rewind()`, `setbuf`, `setvbuf()`

Example This program writes 512 x's to a file and then flushes its buffer.

```
#include <string.h>
#include <stdio.h>

main()
{
   FILE *fp;
   char s[513], c;

   fp = fopen( "MyFile", "w" );

   /* Make s a string of 512 x's */
   memset( s, 'x', 512 );
   s[512] = '\0';

   /* Write them to the file's buffer */
   fputs( s, fp );
```



```
   printf( "If you're using a hard disk,\n"
           "watch the disk light\n"
           "as the stream is flushed.\n\n"
           "Press Return to continue.\n" );
   c = getchar();

   /* Flush the buffer onto the disk. */
   fflush( fp );

   fclose( fp );
}
```

fgetc

Library

ANSI, ANSI-small

Syntax

```
#include <stdio.n>
int fgetc(FILE *stream);
```

Description

To learn how the standard libraries handle streams, see "Using Files and Streams" on page 15.

`fgetc()` reads the next character from the stream `stream`. It returns the integer value of the character in the range 0 to 255.

To read multiple characters more efficiently, use `fread()` or `fgets()`. To read text and data directly into variables, use `scanf()`.

Note

If you open a stream in text mode, `fgetc()` converts Return characters to Line Feeds. For more information, see "Binary and text files" on page 20.

`fgetc()` is similar to `getc()`. `fgetc()` may generate less code, but is less efficient.

Returns

`fgetc()` returns these values:

If the function...	It returns...
succeeds	the next character from the stream.
fails	EOF. If there was a read error, it sets the stream's error indicator. If it reached the end of the file, it sets the stream's EOF indicator.

See Also

`fread()`, `fgets()`, `fscanf()`, `getc()`, `gets()`, `read()`, `scanf()`

Example

This program opens the file `MyFile` and prints its contents to a console window.

```
#include <stdio.h>

main()
{
   FILE *fp;
   int c;

   fp = fopen( "MyFile", "r" );

   while ( (c=fgetc(fp)) != EOF )
      putchar( c );

   fclose( fp );
}
```

fgetpos

Library

ANSI, ANSI-small

Syntax

```
#include <stdio.h>
int fgetpos(FILE *stream, fpos_t *pos);
```

Description

To learn how the standard libraries handle streams, see "Using Files and Streams" on page 15.

`fgetpos()` gets the file position, the point in the file where you'll read from or write to next. It stores in `pos` the file position for the stream `stream`. You can use this value to return you to this position.

`fgetpos()` is similar is `ftell()`. You can't use `ftell()` with huge files (over two billion bytes).

In THINK C, the value in pos is your current offset from the beginning of the file in bytes. However, if your program is portable, use the value only as an argument to `fsetpos()`.

Returns

`fgetpos()` returns these values:

If the function...	It returns...
succeeds	zero.
fails	non-zero.

`fgetpos()` might also set `errno` to this value:

It sets errno to...	If...
ENODEV	`stream` is for a console.

See Also

`fsetpos()`, `fseek()`, `ftell()`, `tell()`

Example

This program opens a file for updating, saves your position, writes a string, restores your old position, and reads in the string.

```
#include <stdio.h>

main()
{
   FILE *fp;
   fpos_t pos;
   char s[80];

   fp = fopen( "MyFile", "w+" );

   fgetpos( fp, &pos );

   fputs( "Hello world.", fp );

   fsetpos( fp, &pos );

   fgets( s, 80, fp );
   printf( "You read: %s\n", s );
   fclose( fp );
}
```

The program writes this line:

```
You read: Hello world.
```

fgets

Library

ANSI, ANSI-small

Syntax

```
#include <stdio.h>
char *fgets(char *s, int n, FILE *stream);
```

Description

To learn how the standard libraries handle streams, see "Using Files and Streams" on page 15.

`fgets()` reads a string. It tries to read the next `n-1` characters from the stream `stream` into the array `s`. Immediately after the last character in `s`, it adds a null character (`'\0'`).

Two things stop `fgets()` from reading `n-1` characters: reading a newline character (`'\n'`) or reaching the end of the file. This is what happens:

- If `fgets()` reads a newline, it adds the newline and a null character (`'\0'`) to `s` and returns successfully.
- If `fgets()` reaches the end of the file after reading some characters, it adds a null character to `s` and returns successfully.
- If `fgets()` reaches the end of the file without reading any characters, sets the EOF error flag for the stream, and returns unsuccessfully. It doesn't alter `s`.

Note

If you open a stream in text mode, `fgets()` converts Return characters to Line Feeds. For more information, see "Binary and text files" on page 20.

Returns

`fgets()` returns the following:

If the function...	It returns...
succeeds	`s`, the string it read.
fails	NULL.

See Also

`fgetc()`, `fputs()`, `getc()`, `getchar()`, `gets()`, `puts()`, `scanf()`, `sscanf()`

Example

This program opens a file for updating, prints a string to it, rewinds it, reads the line back in, and prints the line out.

```
#include <stdio.h>

main()
{
   char buffer[128];
   FILE *fp;

   fp = fopen( "MyFile", "w+" );
   fputs( "Hello world.", fp );
   rewind( fp );

   fgets( buffer, 128, fp );

   printf( "You read: %s", buffer);
   fclose( fp );
}
```

The program prints this line:

```
You read: Hello world.
```

fileno

Not in the ANSI standard.

Library unix

Syntax

```
#include <unix.h>
int fileno(FILE *fp);
```

Description `fileno()` lets you convert ANSI file descriptors to UNIX file numbers. It returns the file number for `fp`, where `fp` is a pointer to a file descriptor.

Returns `fileno()` returns these values:

If the function...	It returns...
succeeds	the file number of the file descriptor.
fails	EOF

See Also `open()`, `close()`, `fdopen()`, `fopen()`, `fclose()`

Example

To learn how the standard libraries handle streams, see "Using Files and Streams" on page 15.

This program opens `MyFile` with `fopen()`, which returns a pointer to a file descriptor. It converts the pointer to a file descriptor to a file number, writes a message to the file, and closes it.

```
#include <stdio.h>
#include <string.h>
#include <unix.h>

main()
{
   FILE *fp;
   int fn;
   char *s = "hello world\r";

   fp = fopen( "MyFile", "w" );

   fn = fileno( fp );

   write( fn, s, strlen(s) );
   close( fn );
}
```


floor

Library ANSI

Syntax

```
#include <math.h>
double floor(double x);
```

Description `floor()` rounds x down to the next lowest integer. It computes the largest integral value not larger than x. For example, `floor(1.1)` is `1.0`, `floor(4.9)` is `4.0`, and `floor(-2.1)` is `-3.0`.

Returns The largest integral value not larger than x, as a double.

See Also `ceil()`, `modf()`

Example This program prints the floor of any number you enter:

```
#include <stdio.h>
#include <math.h>

main()
{
   double x, result;

   printf( "Enter a number: " );
   scanf( "%lf", &x );

   result = floor(x);

   printf( "floor(%G) = %G\n", x, result);
}
```

This is sample output from the program:

```
Enter a number: 2.7
floor(2.7) = 2

Enter a number: -5.4
floor(-5.4) = -6
```

fmod

Library

ANSI

Syntax

```
#include <math.h>
double fmod(double x, double y);
```

Description

`fmod()` computes the floating point remainder of `x/y`. It returns a `double` with the same sign as `x`.

Returns

`fmod()` returns these values:

If the function...	It returns...
succeeds	the remainder of `x/y` with the same sign as `x`.
fails, because `y` is 0	0 (zero). It also sets `errno` to `EDOM`.

See Also

`div()`, `ldiv()`

Example

This program prints the remainder of `x/y` for any `x` and `y` you enter.

```
#include <stdio.h>
#include <math.h>

main()
{
   double x, y, result;

   printf( "Enter x: " );
   scanf( "%lf", &x );

   printf( "Enter y: " );
   scanf( "%lf", &y );

   result = fmod( x, y );

   printf( "The remainder of %G/%G is %G.\n",
           x, y, result );
}
```

This is sample output from the program:

```
Enter x: -8.0
Enter y: 3.0
The remainder of -8/3 is -2.

Enter x: 3.5
Enter y: 0.2
The remainder of 3.5/0.2 is 0.1.
```


fopen

Library

ANSI, ANSI-small

Syntax

```
#include <stdio.h>
FILE *fopen(const char *filename, const char *mode);
```

Description

To learn how the standard libraries handle streams, see "Using Files and Streams" on page 15.

`fopen()` opens a file. It opens the file `filename` according to the mode mode. It automatically allocates a buffer for the file's stream, creates a new stream, and sets the stream to be fully-buffered. It clears the error and EOF indicators for the stream.

Note

`fopen()` opens only the data fork of a file. To open the resource fork, use the Macintosh Toolbox function `OpenRF()`, described in *Inside Macintosh I*, Chapter 5, "Resource Manager."

The three main modes are read, write, and append. Each mode has a set of variations: read binary, write binary, append binary, and so on. These are all the possible modes:

Mode	Meaning
`"r"`	Open this text file for reading.
`"w"`	Create a text file for writing. Overwrite it if it already exists.
`"a"`	Append to the end of this text file.
`"rb"`	Open this binary file for reading.
`"wb"`	Create a binary file for writing. Overwrite it if it already exists.
`"ab"`	Append to the end of this binary file.
`"r+"`	Open this text file for updating (reading and writing).
`"w+"`	Create a text file for updating (reading and writing). Overwrite it if it already exists.
`"a+"`	Open this text file for updating (reading and writing). Append writing to the end.
`"r+b"` or `"rb+"`	Open this binary file for updating.
`"w+b"` or `"wb+"`	Create a binary file for updating. Overwrite it if it already exists.
`"a+b"` or `"ab+"`	Open this binary file for updating. Append writing to the end.

Note

When reading from text files, ANSI functions convert Return characters to Line Feeds. When writing to text files, ANSI functions convert Line Feeds to Returns. For more information, see "Binary and text files" on page 20.

If you try to open a file for reading (mode contains an `r`) that doesn't already exist, `fopen()` fails and returns NULL.

If you open a file for appending (mode contains an `a`), your writing is always appended to the end of the file, even if you call `fseek()` or `fgetpos()`.

`fopen()` lets you open a file for writing or appending on a locked volume. However, you'll get an error if you try to write to that file.

If you open a file for updating (mode contains a `+`), you can both read from and write to the file. However, you must call one of these functions when you want to switch between reading and writing: `fseek()`, `fsetpos()`, `rewind()`. In addition, you can start writing when a read operation reaches the end of the file.

Returns

`fopen()` returns these values:

If the function...	It returns...
succeeds	a pointer to the stream created.
fails	NULL.

`fopen()` might also set `errno` to one of these vaulues:

It sets errno to...	If...
`EINVAL`	mode is not one of the correct values.
`EMFILE`	more than `FOPEN_MAX` (15) files are open.

See Also

`fclose()`, `freopen()`, `rewind()`, `setbuf()`, `setvbuf()`, `tmpfile()`, `tmpnam()`

`open()`, `creat()`, `fileno()`, `fopenc()`, `freopenc()`

Example

This program creates a copy of the file `MyFile`. It opens `MyFile` for reading and `MyFile Backup` for writing. Then it copies `MyFile` to `MyFile Backup`, character by character.

```
#include <stdio.h>

main()
{
   FILE *old, *new;
   int c;

   old = fopen( "MyFile", "rb" );
   new = fopen( "MyFile Backup", "wb" );

   while( (c=fgetc(old)) != EOF )
      fputc( c, new );

   fclose( old );
   fclose( new );
}
```

fopenc

Not in the ANSI standard.

Library

ANSI

Syntax

```
#include <console.h>
FILE *fopenc(void);
```

Description

To learn about console windows, see Chapter 3, "Using Console Windows."

To learn how the standard libraries handle streams, see "Using Files and Streams" on page 15.

Creates a new console.

`fopenc()` creates a new console and opens a new stream on that console.

To set options for the console, set the elements in the global structure, `console_options`. To close a console, use `fclose()`, described in Chapter 3, "ANSI Standard Functions."

Note

Changing `console_options` affects the options for the next console you create. After you create a console, you cannot change its options. For more information , read "Setting console options" on page 44.

Returns

A pointer to a console.

Example

This example creates a console with `fopenc()` and writes to it.

```
#include <console.h>
#include <stdio.h>

main()
{
   FILE *cp;

   console_options.nrows = 12;
   console_options.ncols = 40;
   cp = fopenc();
   fprintf( cp, "hello world\n" );
}
```

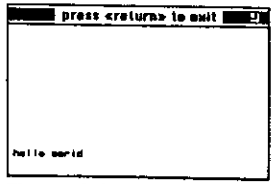

Figure 5-10 Writing to a console created with fopenc().

fprintf

Library

ANSI, ANSI-small

Syntax

```
#include <stdio.h>
int fprintf(FILE *stream, const char *format, ...)
```

Description

To learn how the standard libraries handle streams, see "Using Files and Streams" on page 15.

For complete information on format specifiers, read Chapter 4, "Using printf and scanf."

`fprintf()` writes a formatted string to a stream. It evaluates the data arguments (shown as `...` in the prototype), converts them according to the string `format`, and writes the result to the stream `stream`.

The format string `format` contains ordinary characters and format specifiers. `fprintf()` prints out the ordinary characters directly to the stream. The format specifiers tell it how to print the data arguments. A format specifier begins with a `%` and ends with a conversion character. For example, the format specifier `%d` prints out a decimal integer.

If `printf()` comes to an invalid format specifier, it prints out the converted string up to the specifier, and returns successfully.

Note

If you open a stream in text mode, `fprintf()` converts Line Feed characters to Returns. For more information, see "Binary and text files" on page 20.

Returns

`fprintf()` returns these values:

If the function...	It returns...
succeeds	the number of characters printed.
fails	EOF.

See Also

`fputc()`, `fputs()`, `fscanf()`, `fwrite()`, `printf()`, `putc()`, `puts()`, `sprintf()`, `vfprintf()`, `vprintf()`, `vsprintf()`

Example

This program writes a line to the file `MyFile`.

```
#include <stdio.h>

main()
{
   FILE *fp;

   fp = fopen( "MyFile", "w" );
   fprintf( fp, "pi = %7.5f", 3.141593 );
   fclose( fp );
}
```

After the program runs, `MyFile` contains this line:

```
pi = 3.14159
```


fputc

Library

`ANSI, ANSI-small`

Syntax

```
#include <stdio.h>
int fputc(int c, FILE *stream);
```

Description

For more information on the append mode, see `fopen()` *on page 171. To learn how the standard libraries handle streams, see "Using Files and Streams" on page 15.*

`fputc()` prints the character `c` to the stream `stream`.

`fputc()` writes the character to your current position in the stream, and moves your current position ahead. If you opened the file in append mode, `fputc()` appends the character to the end of the file.

Note

If you open a stream in text mode, `fputc()` converts Line Feed characters to Returns. For more information, see "Binary and text files" on page 20.

Returns

`fputc()` returns these values:

If the function...	It returns...
succeeds	the integer value of the character `c`.
fails	`EOF`.

See Also

`fgetc()`, `fputs()`, `fprintf()`, `fwrite()`, `getc()`, `getchar()`, `putc()`, `putchar()`, `puts()`, `ungetc()`

Example

This program writes the string `"Hello File."` to `MyFile`, character by character.

```
#include <stdio.h>

main()
{
   FILE *fp;
   char *s = "Hello File.";
   int i;

   fp = fopen( "MyFile", "w" );

   for ( i=0; s[i]!='\0'; i++ )
      fputc( s[i], fp );

   fclose( fp );
}
```

fputs

Library

ANSI, ANSI-small

Syntax

```
#include <stdio>
int fputs(const char *s, FILE *stream);
```

Description

To learn how the standard libraries handle streams, see "Using Files and Streams" on page 15.

`fputs()` writes the string `s` to the stream `stream`.

The string must end in a null character (`'\0'`). `fputs()` does not write the null to the stream.

Note

If you open a stream in text mode, `fputs()` converts Line Feed characters to Returns. For more information, see "Binary and text files" on page 20.

Returns

`fputs()` returns these values:

If the function...	It returns...
succeeds	0 (zero)
fails	EOF.

See Also

`fgets()`, `fputc()`, `fprintf()`, `fwrite()`, `gets()`, `putc()`, `putchar()`, `puts()`

Example

This program writes the string `"Hello File."` to `MyFile`.

```
#include <stdio.h>

main()
{
   FILE *fp;
   char *s = "Hello File.";

   fp = fopen( "MyFile", "w" );

   fputs( s, fp );

   fclose( fp );
}
```


fread

Library

ANSI, ANSI-small

Syntax

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t nmemb,
             FILE *stream);
```

Description

To learn how the standard libraries handle streams, see "Using Files and Streams" on page 15.

`fread()` reads a block of data. It tries to read `count` items of size `size` from the stream `stream` and stores them in the array that `ptr` points to.

These are the arguments to `fread()`:

Argument	Description
`ptr`	Pointer to the array that `fread()` stores the items it reads.
`size`	Size of the items that `fread()` reads.
nmemb	Number of items that `fread()` should read.
`stream`	Stream that `fread()` reads from.

If `size` or nmemb is 0 (zero), `fread()` doesn't change the contents of the array.

Note

If you open a stream in text mode, `fread()` converts Return characters to Line Feeds. For more information, see "Binary and text files" on page 20.

Returns

`fread()` always returns the number of items successfully read and stored.

If the function...	It returns...
reaches the end of the stream	the number of items stored.It also sets the stream's EOF indicator.
fails to read count items	the number of items stored.
reads count items	`count`.

See Also

`fgetc()`, `fgets()`, `fwrite()`, `getc()`, `ungetc()`

Example

This code fragment writes out 3 structures of type `Foo` to the file `MyFooFile`, and then reads them back in.

```
#include <stdio.h>

typedef struct {
   int a;
   char b;
} Foo;

main ()
{
   FILE *fp;
   int ;
   Foo inbuf[3];
   static Foo outbuf[3] = { { 1, 'a' },
                            { 2, 'b' },
                            { 3, 'c' } };

   fp = fopen( "MyFooFile", "wb+");

   fwrite( outbuf, sizeof(Foo), 3, fp );

   rewind( fp );
   fread( inbuf, sizeof(Foo), 3, fp );
   for ( i=0; i<3; i++ )
      printf( "foo[%d] = { %d %c }\n",
              i, inbuf[i].a, inbuf[i].b );

   fclose( fp );
}
```

The program writes these lines:

```
foo[0] = { 1 a }
foo[1] = { 2 b }
foo[2] = { 3 c }
```


free

Library

`ANSI, ANSI-small`

Syntax

```
#include <stdlib.h>
void free(void *ptr);
```

Description

To learn how THINK C allocates memory, see "Allocating Memory" on page 20.

`free()` releases a block of memory that `calloc()`, `malloc()`, or `realloc()` allocated. `ptr` is a pointer to the block of memory.

If `ptr` is `NULL`, `free()` returns and does nothing.

Warning

The argument to `free()` must be a pointer that `calloc()`, `malloc()`, or `realloc()` returned. Passing it any other value may crash your program.

See Also

`calloc()`, `malloc()`, `realloc()`

Example

This code fragment allocates and frees a block of memory for an array of structs:

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
   int a, b;
} Foo, FooPtr;

main()
{
   FooPtr foobuf;
   int size = 100;

   foobuf = calloc( sizeof(Foo), size );
   if (foobuf == NULL)
      exit();
   .
   .
   .
   free( foobuf );
}
```

freopen

Library

ANSI, ANSI-small

Syntax

```
#include <stdio.h>
FILE *freopen(const char *filename, const char *mode,
              FILE *stream);
```

Description

To learn how the standard libraries handle streams, see "Using Files and Streams" on page 15.

For more information on mode *see* fopen() *on page 171*

`freopen()` opens a file on a specific stream. It's useful for redirecting the standard streams (`stdin`, `stdout`, and `stderr`) to a file.

These are the arguments to `freopen()`:

Argument	Description
`filename`	the name of the file to open.
`mode`	a string describing how to open the file.
`stream`	the stream to associate with file `filename`.

`freopen()` does the following:

- It closes any file associated with the stream `stream`. If it can't close the file, it continues.
- It opens the file `filename` and associates the stream `stream` with the file.
- It clears the stream's error and EOF flags.

To learn about console windows, see Chapter 3, "Using Console Windows."

If the console package creates a console for the three standard streams (`stdin`, `stdout`, `stderr`), it will close the console after you redirect all three standard streams.

Returns

`freopen()` returns these values:

If the function...	It returns...
succeeds	`stream`.
fails to open the file	`NULL`.

See Also

`fopen()`, `fclose()`, `ccommand()`, `freopenc()`

Example

This example reopens `stdin` as a file, if you directed it to a console in the `ccommand()` dialog

```
#include <unix.h>
#include <stdio.h>
#include <console.h>

main(int argc, char **argv)
{
   char s[80];

   argc = ccommand(&argv);

   if ( (isatty(fileno(stdin))) )
      freopen( "MyFile", "r", stdin );

   gets( s );
   printf( "The first line is:\n%s\n", s );
   fclose( stdin );
}
```

This is sample output:

```
The first line is:
Hello, world.
```

freopenc

Not in the ANSI standard.

Library ANSI

Syntax

```
#include <console.h>
FILE *freopenc(FILE *fp1, FILE *fp2);
```

Description

To learn about console windows, see Chapter 3, "Using Console Windows." To learn how the standard libraries handle streams, see "Using Files and Streams" on page 15.

Opens a console on a specific stream.

`freopenc()` closes the stream that `fp2` refers to, if necessary. It then reopens it as a console as follows:

- If `fp1` is `NULL`, `fp2` is a pointer to a newly-created console.
- If `fp1` refers to a console, `fp2` will refer to that console, too.

Note

Several streams can use one console, as `stdin`, `stdout`, and `stderr` do.

Returns A pointer to the console `fp2`.

Example This example reopens `stdout` as a console, if you directed it to a file in the `ccommand()` dialog.

```
#include <unix.h>
#include <stdio.h>
#include <console.h>

main(int argc, char **argv)
{
   argc = ccommand(&argv);

   if ( !(isatty(fileno(stdout))) )
      freopenc( NULL, stdout );

   printf( "hello world\n" );
}
```


frexp

Library ANSI

Syntax

```
#include <math.h>
double frexp(double value, int *exp);
```

Description `frexp()` splits the floating-point number `value` into two parts:

- Mantissa, a normalized fraction. It's a `double` in the range `0.5` to `1.0`.
- Exponent, a power of two. It's an `int`.

`frexp()` stores the exponent in `*exp` and returns the mantissa. If `value` is 0 (zero), `frexp()` stores 0 in `*exp` and returns 0.

This is how the mantissa and exponent are related to `value`:

$$mantissa * 2^{(exponent)} == value$$

`ldexp()` combines a mantissa and exponent to return a value.

Returns `frexp()` returns these values:

If the function...	It returns...
succeeds	the mantissa. It also stores the exponent in `*exp`.
fails, since `value` is 0 (zero)	0. It also stores 0 in `*exp`.

See Also `exp()`, `ldexp()`, `log()`, `log10()`, `modf()`, `pow()`, `sqrt()`

Example This program splits pi into a mantissa and exponent:

```
#include <stdio.h>
#include <math.h>

main ()
{
   double pi = 3.14159, mantissa;
   int exponent;

   mantissa = frexp( pi, &exponent );
   printf( "%7.5f = %7.5f * 2^%d\n",
           pi, mantissa, exponent );
}
```

The program prints this line:

```
3.14159 = 0.78540 * 2^2
```


fscanf

Library

ANSI, ANSI-small

Syntax

```
#include <stdio.h>
int fscanf(FILE *stream, const char *format, ...);
```

Description

To learn how the standard libraries handle streams, see "Using Files and Streams" on page 15.

`fscanf()` converts a stream's contents into a series of values, using a format string as its guide. Reading from the stream `stream`, it breaks up the input into a series of values according to `format`. It stores these values in the variables that the data arguments point to (shown as ". . ." in the prototype).

For complete information on format specifiers, read Chapter 4, "Using printf and scanf."

The format string `format` contains format specifiers, that tell `fscanf()` how to convert the stream's contents. A format specifier begins with a `%` and ends with a conversion character. For example, `%d` converts some of the stream's contents to an integer. Characters that aren't in a format specifier must match characters from the stream. `fscanf()` doesn't assign them to any data arguments.

`fscanf()` returns unsuccessfully only if it can't read from `stream`. It returns successfully even if it doesn't assign values to all its data arguments. Since it returns the number of data arguments it assigned values to, you can tell whether it assigned to all of them.

Note

If you open a stream in text mode, `fscanf()` converts Return characters to Line Feeds. For more information, see "Binary and text files" on page 20.

Returns

`fscanf()` returns these values:

If the function...	It returns...
converts any input	the number of data arguments it assigned values to.
converts no input	EOF.

See Also

`fgetc()`, `fgets()`, `fprintf()`, `fread()`, `getc()`, `gets()`, `scanf()`, `sscanf()`, `ungetc()`

Example

This program reads some values from the file `MyFile`.

```
#include <stdio.h>

main()
{
    FILE *fp;
    int x;
    char s[80];

    fp = fopen( "MyFile", "r" );
    fscanf( fp, "%d %s", &x, s );
    printf( "x = %d   s = %s\n", x, s );
    fclose( fp );
}
```

If `MyFile` contains this line

```
72 Hello
```

the program prints this line

```
x = 72   s = Hello
```


fseek

Library

`ANSI, ANSI-small`

Syntax

```
include <stdio.h>
int fseek(FILE *stream, long int offset,
          int whence);
```

Description

To learn how the standard libraries handle streams, see "Using Files and Streams" on page 15.

`fseek()` sets the file position for a stream by seeking from the beginning, end, or current file position. The file position is the point in the file where you'll read from or write to next.

For a binary file, `fseek()` sets the file position for stream `stream` to `offset` bytes from `whence`. `whence` can have one of these values:

If whence is...	The function begins from...
`SEEK_SET`	The beginning of the file.
`SEEK_CUR`	The current value of the file position indicator.
`SEEK_END`	The end of the file.

For a text file, one of these must be true:

- `offset` is zero
- `offset` is a value returned by `ftell()` (on `stream`) and `whence` is `SEEK_SET`.

For more information on opening a file in append mode, see `fopen()` on page 171.

You can write only to the end of a file that you've opened in append mode. Calling `fseek()` affects only where you read data and not where you write data.

If `fseek()` is successful, it clears the stream's EOF flag.

Returns

`fseek()` returns these values:

If the function...	It returns...
succeeds	zero.
fails	non-zero.

`fseek()` might also set `errno` to this value:

It sets errno to...	If...
`ENODEV`	`stream` is a console.
`EINVAL`	`pos` is a position beyond the file's limits.

See Also

`fgetpos()`, `fsetpos()`, `ftell()`, `lseek()`

Example

This statement sets the file position to the beginning of a file:

```
fseek( fp, 0L, SEEK_SET );
```

This statement sets the file position to five bytes before the end of the file:

```
fseek( fp, -5L, SEEK_END );
```

If `fp` is a binary file filled with records of type `struct foo`, this statement moves the file position ahead by 10 records:

```
fseek( fp, 10 * sizeof(struct foo), SEEK_CUR );
```


fsetpos

Library

`ANSI, ANSI-small`

Syntax

```
#include <stdio.h>
int fsetpos(FILE *stream, const fpos_t *pos);
```

Description

To learn how the standard libraries handle streams, see "Using Files and Streams" on page 15.

For more information on opening a file in append mode, see `fopen()` *on page 171.*

`fsetpos()` sets the file position for `stream` to `pos`, a value previously returned by `fgetpos()`. The file position is the point in the file where you'll read from or write to next.

You can write only to the end of a file that you've opened in append mode. Calling `fsetpos()` affects only where you read data and not where you write data.

A successful call to `fsetpos()` clears the EOF flag for the stream.

Returns

`fsetpos()` returns these values:

If the function...	It returns...
succeeds	zero
fails	non-zero

`fsetpos()` might also set `errno` to this value:

It sets errno to...	If...
`ENODEV`	`stream` is for a console.
`EINVAL`	`pos` is a position beyond the file's limits.

See Also

`fseek()`, `fgetpos()`, `ftell()`, `lseek()`

Example

This program opens a file for updating, saves your position, writes a string, restores your old position, and reads in the string.

```
#include <stdio.h>

main()
{
   FILE *fp;
   fpos_t pos;
   char s[80];

   fp = fopen( "MyFile", "w+" );

   fgetpos( fp, &pos );

   fputs( "Hello world.", fp );

   fsetpos( fp, &pos );

   fgets( s, 80, fp );
   printf( "You read: %s\n", s );
   fclose( fp );
}
```


ftell

Library

ANSI, ANSI-small

Syntax

```
#include <stdio.h>
long int ftell(FILE *stream);
```

Description

To learn how the standard libraries handle streams, see "Using Files and Streams" on page 15.

`ftell()` returns the current file position for the stream `stream`. The file position is the point in the file where you'll read from or write to next.

The `fseek()` function can use the return value to bring the file position to its location at the time of the `ftell()` call.

For binary files, `ftell()` returns the offset from the beginning of the file in bytes. For text files, its return value isn't necessarily useful, except as an argument to `fseek()`.

`fgetpos()` is similar to `ftell()`. `fgetpos()` is especially useful for huge files (over 2 billion bytes).

Returns

`ftell()` returns these values:

If the function...	It returns...
succeeds	the current file position.
fails	`-1L`.

`ftell()` might also set `errno` to this value:

It sets errno to...	If...
ENODEV	`stream` is for a console.

See Also

`fgetpos()`, `fsetpos()`, `fseek()`, `tell()`

Example

This program opens a file for updating, saves your position, writes a string, restores your old position, and reads in the string.

```
#include <stdio.h>

main()
{
   FILE *fp;
   long int pos;
   char s[80];

   fp = fopen( "MyFile", "w+" );

   pos = ftell( fp );

   fputs( "Hello world.", fp );

   fseek( fp, pos, SEEK_SET );

   fgets( s, 80, fp );
   printf( "You read: %s\n", s );
   fclose( fp );
}
```

The program writes this line:

```
You read: Hello world.
```


fwrite

Library

ANSI, ANSI-small

Syntax

```
#include <stdio.h>
size_t fwrite(const void *ptr, size_t size,
              size_t nmemb, FILE *stream);
```

Description

To learn how the standard libraries handle streams, see "Using Files and Streams" on page 15.

`fwrite()` writes a block of data to stream. It writes to the stream `stream` nmemb items of size `size` from the array that `ptr` points to.

These are the arguments to `fwrite()`:

Argument	Description
`ptr`	Pointer to the memory to write to.
`size`	Size (uniform) of the items to write.
nmemb	Number of items to write.
`stream`	Stream to write to.

Note

If you open a stream in text mode, `fwrite()` converts Line Feed characters to Returns. For more information, see "Binary and text files" on page 20.

Returns

`fwrite()` always returns the number of items successfully written:

If the function...	It returns...
fails to write count items	the number of items written.
writes `count` items	`count`.

See Also

`fprintf()`, `fputc()`, `fputs()`, `fread()`, `putc()`, `puts()`, `ungetc()`, `write()`

Example

This code fragment writes out 3 items of the structure `Foo` to the file `MyFooFile`, and then reads them back in.

```
#include <stdio.h>

typedef struct {
   int a;
   char b;
} Foo;

main ()
{
   FILE *fp;
   int i;
   Foo inbuf[3];
   Foo outbuf[3] = { { 1, 'a' },
                     { 2, 'b' },
                     { 3, 'c' } };

   fp = fopen( "MyFooFile", "wb+");
   fwrite( outbuf, sizeof(Foo), 3, fp );
   rewind( fp );

   fread( inbuf, sizeof(Foo), 3, fp );

   for ( i=0; i<3; i++ )
      printf( "foo[%d] = { %d %c }\n",
              i, inbuf[i].a, inbuf[i].b );

   fclose( fp );
}
```

The program writes these lines:

```
foo[0] = { 1 a }
foo[1] = { 2 b }
foo[2] = { 3 c }
```


getc

Library

`ANSI, ANSI-small`

Syntax

```
#include <stdio.h>
int getc(FILE *stream)
```

Description

To learn how the standard libraries handle streams, see "Using Files and Streams" on page 15.

`getc()` reads the next character from the stream `stream` and returns the integer value of the character in the range 0 to 255.

To read multiple characters more efficiently, use `fread()` or `gets()`. To read text and data directly into variables, use `scanf()`.

`getc()` is similar to `fgetc()`. `getc()` is more efficient, but may generate more code.

`getc()` may evaluate `stream` more than once. Make sure `stream` is not an expression that can have side effects.

Note

If you open a stream in text mode, `getc()` converts Return characters to Line Feeds. For more information, see "Binary and text files" on page 20.

Note

When you're using the console package environment, the mode you set with `csetmode()` can change how `getc()` behaves. For more information, see "Setting the input mode" on page 45.

Returns

`fgetc()` returns these values:

If the function...	It returns...
succeeds	the next character from the stream.
fails	EOF. If there was a read error, it sets the stream's error indicator. If it reached the end of the file, it sets the stream's EOF indicator.

See Also

`fgetc()`, `fgets()`, `fread()`, `fscanf()`, `getchar()`, `gets()`, `ungetc()`

Example

This program prints the file `MyFile` to standard output.

```
#include <stdio.h>

main()
{
   FILE *fp;
   int c;

   fp = fopen( "MyFile", "r" );

   while ( (c=getc(fp)) != EOF )
      putchar( c );

   fclose( fp );
}
```


getchar

Library

`ANSI, ANSI-small`

Syntax

```
#include <stdio.h>
int getchar(void)
```

Description

To learn how the standard libraries handle streams, see "Using Files and Streams" on page 15.

`getchar()` reads the next character from standard input and returns the integer value of the character in the range 0 to 255.

To read multiple characters more efficiently, use `fread()` or `gets()`. To read text and data directly into variables, use `scanf()`.

`getchar()` is similar to `fgetc()`. `getchar()` is more efficient, but may generate more code. Also, `fgetc()` requires a stream as an arugment.

Note

When you're using the console package environment, the mode you set with `csetmode()` can change how `getchar()` behaves. For more information, see "Setting the input mode" on page 45.

Returns

`fgetc()` returns these values:

If the function...	It returns...
succeeds	the next character from the stream.
fails	EOF. If there was a read error, it sets the standard input's error indicator. If it read an EOF character, it sets standard input's EOF indicator.

See Also

`fgetc()`, `fgets()`, `fputc()`, `getc()`, `gets()`, `putc()`, `putchar()`, `scanf()`, `ungetc()`

Example

This program reads in a character:

```
#include <stdio.h>

main()
{
   char c;

   printf( "Type a character: " );

   c = getchar();

   printf( "You typed '%c'.\n", c );
}
```


getegid
geteuid
getgid
getpgrp
getpid
getppid
getuid

Not in the ANSI standard.

Library

unix

Syntax

```
#include <unix.h>
int getegid(void);
int geteuid(void);
int getgid(void);
int getpgrp(void);
int getpid(void);
int getppid(void);
int getuid(void);
```

Description

In UNIX, these functions return ID numbers for a process. In THINK C, these return fixed numbers that represent possible values for processes on a UNIX system.

Function	Value	Description
`getegid()`	31	Group ID
`geteuid()`	24	User ID
`getgid()`	31	Process's group ID
`getpgrp()`	57	Process group ID of the calling process
`getpid()`	57	Program's process ID number
`getppid()`	5	Calling process's parent's process ID
`getuid()`	24	Program's user ID number

getenv

Library

ANSI

Syntax

```
#include <stdio.h>
char *getenv(const char *name);
```

Description

`getenv()` always returns NULL. It's provided for ANSI and UNIX compatibility.

To get information about the Macintosh your program is running on, use the Gestalt Manager, described in *Inside Macintosh VI*, Chapter 3, "Compatibility Guidelines."

Returns

NULL

getlogin

Not in the ANSI standard.

Library

unix

Syntax

```
#include <unix.h>
char *getlogin(void);
```

Description

`getlogin()` returns the user name set in the Chooser desk accessory. In the UNIX environment, `getlogin()` returns the username under which the program was started.

Returns

The user name set in the Chooser desk accessory.

Example

This program greets you, using the name from the Chooser:

```
#include <stdio.h>
#include <unix.h>

main()
{
   char *name;

   name = getlogin();

   printf( "Hello, %s.\n", name );
}
```

This is sample output from the program:

```
Hello, Fred.
```

gets

Library

ANSI, ANSI-small

Syntax

```
#include <stdio.h>
char *gets(char *s);
```

Description

To learn how the standard libraries handle streams, see "Using Files and Streams" on page 15.

`gets()` reads a line from standard input and stores it in `s`.

`gets()` stops reading when it reads a newline or EOF character. It appends a null character to `s` and returns. It doesn't put the newline or EOF character in `s`.

If it reads an EOF character before it reads any other characters, it sets standard input's EOF flag and returns unsuccessfully without changing `s`.

Returns

`gets()` returns these values:

If the function...	It returns...
succeeds	`s`.
fails	NULL.

See Also

`fgetc()`, `fgets()`, `fputs()`, `getc()`, `getchar()`, `puts()`, `ungetc()`

Example

This program asks for your name and greets you:

```
#include <stdio.h>

main ()
{
   char name[80];

   printf( "What's your name? ");

   gets( name );

   printf( "Hello, %s.\n", name );
}
```


getw

Not in the ANSI standard.

Library

unix

Syntax

```
#include <unix.h>
int getw(FILE *stream);
```

Description

To learn how the standard libraries handle streams, see "Using Files and Streams" on page 15.

`getw()` returns the next two bytes of `stream` as an integer, where `stream` is a pointer to a file descriptor.

`getw()` assumes the bytes are in memory order. On the MC68000 the most significant byte is in the first byte.

Returns

`getw()` returns these values:

If the function...	It returns...
succeeds	An integer containing the next two bytes of the file.
fails	EOF

See Also

`putw()`, `read()`, `fread()`

Example

This program opens the file `MyFile`, reads the first two bytes, and writes them out as characters.

```
#include <stdio.h>
#include <unix.h>

main()
{
   FILE *fp;
   int w;

   fp = fopen( "MyFile", "rb" );

   w = getw( fp );

   printf( "You read: %.2s\n", &w );
   fclose( fp );
}
```

If `MyFile` contains this line:

```
hello world.
```

the program writes this:

```
You read: he
```

gmtime

Library
ANSI

Syntax
```
#include <time.h>
struct tm *gmtime(const time_t *timer);
```

Description
gmtime() always returns NULL.

gmtime() is provided for ANSI and UNIX compatibility only. According to the standard, it converts the calendar time in `timer` to Coordinated Universal Time (UTC).

Returns
NULL

See Also
ctime(), localtime(), time()

isatty

Not in the ANSI standard.

Library
unix

Syntax
```
#include <unix.h>
int isatty(int fn);
```

Description
To learn about console windows, see Chapter 3, "Using Console Windows."

`isatty()` tests whether the stream associated with file number `fn` is for a console. Use it with `ccommand()`, on page 122, to see if the user redirected the standard streams.

Returns
`isatty()` returns these values:

If the stream is...	The function returns...
for a console	Non-zero
not for a console	Zero

See Also
ccommand()

Example
This program determines if the user redirected `stdout` in the `ccommand()` dialog.

```
#include <stdio.h>
#include <console.h>
#include <unix.h>

main(int argc, char **argv)
{
   argc = ccommand(&argv);

   if ( isatty(fileno(stdout)) )
      printf("Whew, someone's out there.\n");
   else
      printf("I'm talking to a file.\n");
}
```

isxxxx

Library

ANSI, ANSI-small, ANSI-A4

Syntax

```
#include <ctype.h>
int isxxx(int c);
```

Description

isxxx() represents a family of functions that let you determine what category a character belongs to: printable characters, control characters, digits, lower-case letters, and so on.

Each of these functions takes a single character c as an argument. It returns 1 if c is a member of the category, and a non-zero value if c isn't.

This function...	Tests whether c is...
isalnum()	an alphanumeric character (a-z, A-Z, 0-9).
isalpha()	an alphabetic character (a-z, A-Z).
iscntrl()	a control character (ASCII codes 0x0-0x1F inclusive and 0x7F).
isdigit()	a decimal digit (0-9).
isgraph()	a printing character, but not a space (' ') (ASCII codes 0x21-0x7E inclusive).
islower()	a lowercase character (a-z).
isprint()	a printing character, including space (' ') (ASCII codes 0x20-0x7E inclusive except 7F).
ispunct()	a printing character that is neither a space (' ') nor a character for which isalnum() is true. (See the list below.)
isspace()	a standard white-space character. (See the list below)
isupper()	an uppercase character (A-Z).
isxdigit()	a hexadecimal digit (0-9, A-F, a-f).

These are the punctuation characters, for ispunct():

```
# $ % * ( ) " ' \ ` ~ : ; ,
< > / ? \ | [ ] { } - _ = +
```


These are the whitespace characters, for isspace():

- space (' ')
- form feed ('\f')
- new-line ('\n')
- carriage return ('\r')
- horizontal tab ('\t')
- vertical tab ('\v')

Returns

These functions return these values:

If c is...	The function returns...
in the category	0 (zero).
not in the category.	non-zero.

Example

This program opens the binary file MyData and prints its contents, replacing unprintable characters with a dot ('•').

```
#include <stdio.h>
#include <ctype.h>

main ()
{
    int c;
    FILE *fp;

    fp = fopen( "MyData", "rb" );

    while ( (c=fgetc(fp)) != EOF ) {
        if ( isprint(c) )
            putchar(c);
        else
            putchar('•');
    }

    fclose(fp);
}
```

kill

Not in the ANSI standard.

Library

unix

Syntax

```
#include <unix.h>
int kill(int pid, int sig);
```

Description

kill() raises the signal sig to your program.

kill() raises the signal sig if sig is not zero and if pid is one of the following:

- 0
- -1
- the value returned by getpid()
- the negative of the value returned by getpgrp().

If pid is not one of those values, kill() returns -1 and sets errno to ESRCH.

kill(pid, sig) is equivalent to this statement, if pid is one of the proper values:

```
raise( sig );
```

Under UNIX, kill() sends the signal to the process specified by pid. In THINK C, the unix library doesn't support inter-application communication, so pid must be a value that represents the currently running process.

Returns

kill() returns these values:

If the function...	It returns...
succeeds	0
fails	-1. It also sets errno to ESRCH.

See Also

signal(), raise()

labs

Library

ANSI, ANSI-small, ANSI-A4

Syntax

```
#include <stdlib.h>
long int labs(long int j);
```

Description

labs() returns the absolute value of the long integer j.

Returns

| j | for any integer j, as a long integer.

See Also

abs(), fabs()

Example

This program prints the absolute value of any long integer number you enter:

```
#include <stdio.h>
#include <stdlib.h>

main()
{
   long int x, result;

   printf( "Enter a number: " );
   scanf( "%ld", &x );

   result = labs(x);

   printf( "labs(%ld) = %ld.\n", x, result);
}
```

This is sample output from the program:

```
Enter a number: -823790539
labs(-823790539) = 823790539.
```

ldexp

Library ANSI

Syntax

```
#include <math.h>
double ldexp(double x, int exp);
```

Description `ldexp()` returns x times 2 to the power of exp ($x * 2^{exp}$).

`frexp()` is the inverse function of `ldexp()`. `frexp()` splits a number into two parts, x and exp.

Returns `ldexp()` returns these values:

If the function...	It returns...
succeeds	$x * 2^{exp}$
fails, since x is too big	Infinity. It also sets `errno` to ERANGE.

See Also `exp()`, `frexp()`, `log()`, `log10()`, `modf()`, `pow()`, `sqrt()`

Example This program:

```
#include <stdio.h>
#include <math.h>

main ()
{
   double x = 0.78540;
   int exp = 2;

   printf( "%7.5f * 2^%d = %7.5f\n",
           x, exp, ldexp(x, exp) );
}
```

prints this line:

```
0.78540 * 2^2 = 3.14160
```


ldiv

Library ANSI, ANSI-small, ANSI-A4

Syntax

```
#include <stdlib.h>
ldiv_t ldiv(long int numer, long int denom);
```

Description `ldiv()` returns, as long integers, the quotient and remainder for the division of `numer` by `denom`. It returns a structure of type `ldiv_t`, which has these fields:

Field	Type	Description
`quot`	`long int`	The quotient
`rem`	`long int`	The remainder

This is how `quot` and `rem` are related to `numer` and `denom`:

```
quot * denom + rem == numer.
```

`ldiv()` is similar to `div()`. `ldiv()` takes `long ints` as arguments and returns `long ints`. `div()` takes `ints` as arguments and returns `ints`.

Returns A structure with the results of the division.

See Also `div()`, `fmod()`

Example This program finds the quotient and remainder for `239774283 / 8237482`.

```
#include <stdio.h>
#include <stdlib.h>

main ()
{
   long int n = 239774283, d = 8237482;
   ldiv_t q;

   q = ldiv(n,d);
   printf( "%ld / %ld is %ld\n", n, d, q.quot );
   printf( "with a remainder of %ld.\n", q.rem);
}
```

The program prints these lines:

```
239774283 / 8237482 is 29
with a remainder of 887305.
```

localeconv

Library

ANSI

Syntax

```
#include <locale.h>
struct lconv *localeconv(void);
```

Description

`localeconv()` is provided for ANSI C compatibility only. THINK C supports only the minimal "C" environment.

According to the standard, `localeconv()` returns an object of type `struct lconv`, with values appropriate for the formatting of numeric quantities (monetary and otherwise) according to the rules of the current locale. In THINK C, `localeconv()` returns values for the "C" locale, the minimal environment.

`struct lconv` includes these fields:

Field	Type	Value in "C" locale
decimal_point	char *	"."
thousands_sep	char *	""
grouping	char *	""
int_curr_symbol	char *	""
currency_symbol	char *	""
mon_decimal_point	char *	""
mon_thousands_sep	char *	""
mon_grouping	char *	""
positive_sign	char *	""
negative_sign	char *	""
int_frac_digits	char	CHAR_MAX
frac_digits	char	CHAR_MAX
p_cs_precedes	char	CHAR_MAX
p_sep_by_space	char	CHAR_MAX
n_cs_precedes	char	CHAR_MAX
n_sep_by_space	char	CHAR_MAX
p_sign_posn	char	CHAR_MAX
n_sign_posn	char	CHAR_MAX

To localize your software for other countries, use the International Utilities Package, described in *Inside Macintosh VI*, Chapter 14, "Worldwide Software Overview."

Returns

A pointer to an object of type `struct lconv`.

localtime

Library

ANSI

Syntax

```
#include <time.h>
struct tm *localtime(const time_t *timer);
```

Description

`localtime()` breaks a calendar time into its constituent parts. It takes the calendar time that `timer` points and converts it to a broken-down time.

In THINK C, the calendar time is the number of seconds since midnight, January 1, 1904. The broken-down time is of type `struct tm`, and includes these fields: (They're all of type `int`.)

Field	Description
tm_sec	Seconds after the minute
tm_min	Minutes after the hour
tm_hour	Hours since midnight
tm_mday	Day of the month
tm_mon	Months since January
tm_year	Years since 1900
tm_wday	Days since Sunday
tm_yday	Days since January 1
tm_isdst	Daylight Savings Time flag

Returns

A pointer to the broken-down time.

See Also

`asctime()`, `mktime()`, `strftime()`, `time()`

Example

This program prints the current date and time:

```
#include <stdio.h>
#include <time.h>

main()
{
   time_t now;
   struct tm *date;
   char *s;

   now = time( NULL );
   date = localtime( &now );
   s = asctime( date );
   printf( "%s", s );
}
```

log

Library ANSI

Syntax

```
#include <math.h>
double log(double x)
```

Description `log()` returns the natural logarithm of x.

Returns `log()` returns these values:

If x is...	This function returns...
greater than 0 (zero)	ln x, the natural logarithm of x.
0 (zero) or negative	Negative infinity. It also sets `errno` to EDOM.

See Also `exp()`, `log10()`, `pow()`, `sqrt()`

Example This program prints the natural logarithm for any number you enter:

```
#include <stdio.h>
#include <math.h>

main()
{
   double x, result;

   printf( "Enter a number: " );
   scanf( "%lf", &x );

   result = log(x);

   printf( "log(%G) = %G\n", x, result);
}
```

This is sample output from the program:

```
Enter a number: 3.0
log(3) = 1.09861
```


log10

Library ANSI

Syntax

```
#include <math.h>
double log10(double x)
```

Description `log10()` returns the base-10 logarithm of x.

Returns `log10()` returns these values:

If x is...	This function returns...
greater than 0 (zero)	log x, the base-10 logarithm of x.
0 (zero) or negative	Negative infinity. It also sets `errno` to EDOM.

See Also `exp()`, `log()`, `pow()`, `sqrt()`

Example This program prints the base-10 logarithm for any number you enter:

```
#include <stdio.h>
#include <math.h>

main()
{
   double x, result;

   printf( "Enter a number: " );
   scanf( "%lf", &x );

   result = log10(x);

   printf( "log10(%G) = %G\n", x, result);
}
```

This is sample output from the program:

```
Enter a number: 3.0
log10(3) = 0.477121
```

lseek

Not in the ANSI standard.

Library

unix

Syntax

```
#include <unix.h>
long lseek(int fn, long offset, int whence);
```

Description

To learn how the standard libraries handle streams, see "Using Files and Streams" on page 15.

`lseek()` sets the file position for a stream, by seeking from the beginning, end, or current file position. It is identical to `fseek()`, except that `lseek()` takes a file number instead of a file descriptor.

For a binary file, `lseek()` sets the file position for file `fn` to `offset` bytes from `whence`. `whence` can have one of these values:

Macro	Description
SEEK_SET	The beginning of the file.
SEEK_CUR	The current value of the file position indicator.
SEEK_END	The end of the file.

For a text file, one of these must be true:

- `offset` is zero
- `offset` is a value returned by `tell()` (on stream `stream`) and `whence` is `SEEK_SET`.

`lseek()` is useful for performing nonsequential I/O to a file since it lets you manipulate the file position directly. Otherwise, you have to use other functions, like `read()`, throw out what they return, and follow the file position through the file.

If `lseek()` is successful, it clears the stream's EOF flag.

Returns

`lseek()` returns these values:

If the function...	It returns...
succeeds	the value of the file position indicator
fails	EOF

See Also

`tell()`, `fseek()`

Example

This statement sets the file position to the beginning of a file:

```
lseek( fn, 0L, SEEK_SET );
```

This statement sets the file position to five bytes before the end of the file:

```
lseek( fn, -5L, SEEK_END );
```


If `fp` is a binary file filled with records of type `struct foo`, this statement moves the file position ahead by 10 records:

```
lseek( fn, 10 * sizeof(struct foo), SEEK_CUR );
```

longjmp

Library

ANSI, ANSI-small

Syntax

```
#include <setjmp.h>
void longjmp(jmp_buf env, int val);
```

Description

`longjmp()`, together with `setjmp()`, lets you jump back to a function earlier in the subroutine call chain.

`longjmp()` restores the environment that was saved by a previous `setjmp()` call, using the information stored in `env`. Your program continues running as if the `setjmp()` call just finished and returned the value `val`.

Don't use `longjmp()` if you haven't called `setjmp()` or if you called `setjmp()` in a function that has since returned.

These are the arguments to `longjmp()`:

Argument	Description
`env`	The environment to restore. It must be the environment that a previous `setjmp()` call provided.
`val`	The value that the `setjmp()` call returns. If it's 0 (zero), `longjmp()` replaces it with 1.

After you call `longjmp()`, some local variables may not have the right values. These variables are those local to the function that called `setjmp()` and that changed value between the calls to `setjmp()` and `longjmp()`. To make sure that these variables have the correct values, declare them `volatile`. In this example below, `i` must be declared `volatile` to ensure that it has the right value.

Note

You can't use `setjmp()` and `longjmp()` with object methods.

See Also

`setjmp()`

Example

This program shows you how to use `setjmp()` and `longjmp()`. Note that `i` must be declared `volatile` to have the right value.

```
#include <stdio.h>
#include <setjmp.h>

void foo(void);
jmp_buf env;

main ()
{
   volatile int i = 9;

   if ( setjmp(env) == 0 )
      printf( "Called setjmp() to set up a "
              "return point. i = %d\n", i);
   else {
      printf( "Returned from longjmp(). "
              "i = %d\n", i);
      exit();
   }

   i = 10;
   foo();
}

void foo(void)
{
   printf( "Now in foo().\n");
   longjmp(env, 1);
   printf( "This line won't be printed.\n" );
}
```

This is what the program prints:

```
Called setjmp() to set up a return point. i = 9
Now in foo().
Returned from longjmp(). i = 10
```

malloc

Library

ANSI, ANSI-small

Syntax

```
#include <stdlib.h>
void *malloc(size_t size);
```

Description

To learn how THINK C allocates memory, see "Allocating Memory" on page 20.

`malloc()` allocates a block of memory for an object of size `size`.

Unlike `calloc()`, `malloc()` does not initialize the block of memory.

To deallocate the block of memory, use `free()`.

Warning

If you have any problems with `malloc()`, make sure you #included `stdlib.h` in your source file.

Returns

`malloc()` returns these values:

If the function...	It returns...
allocated the block	a pointer to the beginning of the allocated block of memory
didn't allocate the block	NULL

See Also

`free()`, `calloc()`, `realloc()`

Example

This code fragment allocates a block of memory for an array of structs:

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
   int a, b;
} Foo, *FooPtr;

main()
{
  FooPtr foobuf;
  int size = 100;

  foobuf = (FooPtr) malloc( size*sizeof(Foo) );
  if (foobuf == NULL)
    exit();

  foobuf[0].a = 12;
  foobuf[30].b = 100;
  . . .
}
```


mblen

Library

ANSI

Syntax

```
#include <stdlib.h>
int mblen(const char *s, size_t n);
```

Description

`mblen()` returns 0 if `s` is NULL, 1 otherwise.

`mblen()` is provided for ANSI C compatibility only, since THINK C does not support multi-byte characters.According to the ANSI standard, it returns the number of bytes in the multi-byte character that `s` points to.

To use characters that are more than one-byte long, use the Script Manager, described in *Inside Macintosh VI*, Chapter 14, "Worldwide Software Overview.

Returns

`mblen()` returns these values

If s is...	This function returns...
not NULL	1.
NULL	0 (zero).

mbstowcs

Library ANSI

Syntax

```
#include <stdlib.h>
size_t mbstowcs(wchar_t *pwcs, const char *s,
                size_t n);
```

Description

`mbstowcs()` copies the elements of `s` to `pcws`, stopping after it copies `n` elements or a null (`'\0'`) character.

`mbstowcs()` is provided for ANSI C compatibility only, since THINK C does not support multi-byte characters. According to the ANSI standard, it converts a multi-byte character `s` to a sequence of codes, stores it in the array `pcws`, and returns the number of array elements changed.

To use characters that are more than one-byte long, use the Script Manager, described in *Inside Macintosh VI*, Chapter 14, "Worldwide Software Overview.

Returns The number of characters copied.

mbtowc

Library ANSI

Syntax

```
#include <stdlib.h>
int mbtowc(wchar_t *pwc, const char *s, size_t n);
```

Description

`mbtowc()` sets `pwc` to the first element of `s`, if `s` is not null (`'\0'`) and `n` is not 0.

`mbtowc()` is provided for ANSI C compatibility only, since THINK C does not support multi-byte characters.According to the ANSI standard, it converts the multi-byte `s` to a code (converting no more than `n` bytes), stores the code in the object `pwc` points to, and returns the length in bytes of the multi-byte character.

To use characters that are more than one-byte long, use the Script Manager, described in *Inside Macintosh VI*, Chapter 14, "Worldwide Software Overview.

Returns `mbtowc()` returns these values:

If...	The function returns...
the function succeeds	`1`.
`s` is `NULL`	0 (zero).
`n` is 0 (zero)	`-1`.

memchr

Library

ANSI, ANSI-small, ANSI-A4

Syntax

```
#include <string.h>
void *memchr(const void *s, int c, size_t n);
```

Description

memchr() looks for the character c in the block of memory s. It returns a pointer to first occurrence of c in the initial n characters of the object s.

Make sure that c is between 0 and 255. memchr() converts it to an unsigned char.

To search for a character in a null-terminated string, try strchr().

Returns

memchr() returns these values:

If the function...	It returns...
finds the character	a pointer to the character in s.
doesn't find it	NULL.

See Also

strchr(), strcspn(), strpbrk(), strrchr(), strspn(), strstr()

Example

This program finds the first t in "Symantec".

```
#include <stdio.h>
#include <string.h>

main()
{
   char *s = "Symantec";
   char c = 't';

   printf( "s = \"%s\"\n", s );
   printf( "c = '%c'\n", c );
   printf( "memchr(s,c,8) = \"%s\"\n",
           memchr(s,c,8) );
}
```

The program prints these lines:

```
s = "Symantec"
c = 't'
memchr(s,c,8) = "tec"
```


memcmp

Library

ANSI, ANSI-small, ANSI-A4

Syntax

```
#include <string.h>
int memcmp(const void *s1, const void *s2,
           size_t n);
```

Description

memcmp() compares two blocks of memory. It compares the first n characters of the object s1 to the first n characters of the object s2.

To compare two null-terminated strings, try strcmp() or strncmp().

Returns

memcmp() returns these values:

If s1 is...	The function returns...
greater than s2	a positive integer.
equal to s2	0 (zero)
less than s2	a negative integer.

See Also

strcmp(), strncmp()

Example

This program lets you enter values for two structures and compares them:

```
#include <stdio.h>
#include <string.h>

typedef struct {
   int a;
   int b;
} Foo;

main()
{
   Foo foo1, foo2;

   printf( "For each structure, enter two\n"
           "integers separated by a space,\n"
           "like this: 123 987\n\n" );

   printf( "Enter two integers for foo1: ");
   scanf( "%d %d", &foo1.a, &foo1.b );
   printf( "Enter two integers for foo2: ");
   scanf( "%d %d", &foo2.a, &foo2.b );

   if ( memcmp(&foo1, &foo2, sizeof(Foo)) == 0 )
     printf( "foo1 and foo2 are the same.\n" );
   else
     printf( "foo1 and foo2 are different.\n" );
}
```

memcpy

Library ANSI, ANSI-small, ANSI-A4

Syntax

```
#include <string.h>
void *memcpy(void *s1, const void *s2, size_t n);
```

Description memcpy() copies characters from one block of memory to another. It copies n characters from the object s2 into the object s1.

If the two objects might overlap, try memmove(), since memcpy() may destroy data. If you want to copy a null-terminated string, try strcpy() or strncpy().

Returns s1, after copying the first n characters of s2 into it.

See Also memmove(), strcpy(), strncpy()

Example This program copies the contents of one structure to another.

```
#include <stdio.h>
#include <string.h>

typedef struct {
   int a;
   int b;
} Foo;

main()
{
   Foo foo1 = { 123, 456 };
   Foo foo2 = { 987, 321 };

   printf( "Before memcpy():\n"
           " foo1 = %d, %d   foo2 = %d, %d\n",
           foo1.a, foo1.b, foo2.a, foo2.b );

   memcpy( &foo1, &foo2, sizeof(Foo) );

   printf( "After memcpy():\n"
           " foo1 = %d, %d   foo2 = %d, %d\n",
           foo1.a, foo1.b, foo2.a, foo2.b );
}
```

The program prints these lines:

```
Before memcpy():
 foo1 = 123, 456   foo2 = 987, 321
After memcpy():
 foo1 = 987, 321   foo2 = 987, 321
```


memmove

Library ANSI, ANSI-small, ANSI-A4

Syntax

```
#include <string.h>
void *memmove(void *s1, const void *s2, size_t n);
```

Description memmove() copies characters from one block of memory to another, even if the two blocks overlap. It copies n character from the object s2 into the object s1.

memmove() works correctly even if s1 and s2 overlap. If you know the two objects don't overlap, use memcpy() instead, since it's faster. If you want to copy a null-terminated string, try strcpy() or strncpy().

Returns s1, after copying the first n characters of s2 into it.

See Also memcpy(), strcpy(), strncpy()

Example This program:

```
#include <stdio.h>
#include <string.h>

main()
{
   char s[16] = "xxx ooo XXX OOO";

   printf( "Before memmove(): s = %s\n", s );
   memmove( &s[4], &s[0], 7 );
   printf( "After memmove():  s = %s\n", s );
}
```

prints this:

```
Before memmove(): s = xxx ooo XXX OOO
After memmove():  s = xxx xxx ooo OOO
```

memset

Library

ANSI, ANSI-small, ANSI-A4

Syntax

```
#include <string.h>
void *memset(void *s, int c, size_t n);
```

Description

memset() initializes a block a memory. It copies the value of c into each of the first n characters of the object s.

Make sure c is in the range 0 to 255. memset() coverts it to an unsigned char.

Returns

s, after setting the first n characters of it to c.

See Also

calloc()

Example

This program:

```
#include <stdio.h>
#include <string.h>

main()
{
    char s[12] = "Hello world";

    printf( "Before memset(): s = \"%s\"\n", s);

    memset( s, ' ', 11 );

    printf( "After memset():  s = \"%s\"\n", s);
}
```

prints this:

```
Before memset(): s = "Hello world"
After memset():  s = "           "
```


mktime

Library

ANSI

Syntax

```
#include <time.h>
time_t mktime(struct tm *timeptr);
```

Description

mktime() converts the broken-down time in timeptr into a calendar time. You can also use it to:

- find the day of the week for a certain date
- find the number of days since the beginning of the year for a certain date
- force all the fields in a broken-down time structure to their proper ranges.

Calendar time is an integer representing the date and time. In THINK C it's the number of seconds since midnight, January 1, 1904. Broken-down time is a structure representing the date and time. It includes these fields: (They're all of type int.)

Field	Description (range)
tm_sec	Seconds after the minute (0 to 61)
tm_min	Minutes after the hour (0 to 59)
tm_hour	Hours since midnight (0 to 23)
tm_mday	Day of the month (1 to 31)
tm_mon	Months since January (0 to 11)
tm_year	Years since 1900
tm_wday	Days since Sunday (0 to 6)
tm_yday	Days since January 1 (0 to 365)
tm_isdst	Daylight Savings Time flag

mktime() fills tm_wday and tm_yday with the correct values, ignoring the original values. It also adjusts the other fields if they're out of range.

Returns

mktime() returns these values:

If the function...	It returns...
succeeds	the calendar time.
can't represent the time	(time_t) -1.

See Also

asctime(), ctime(), difftime(), localtime(), strftime(), time()

Example

This figures the day of the week for January 1, 2000:

```
#include <stdio.h>
#include <time.h>

char *dname[] = { "Sunday", "Monday", "Tuesday",
                  "Wednesday", "Thursday",
                  "Friday", "Saturday" };

main()
{
   struct tm date;

   /*
    * Set up date for Midnight, January 1, 2000.
    */
   date.tm_sec = 0;
   date.tm_min = 0;
   date.tm_hour = 0;
   date.tm_mday = 1;
   date.tm_mon = 0;
   date.tm_year = 100;
   date.tm_isdst = -1;

   if ( mktime(&date)  == -1 )
      printf( "OOPS, invalid date.\n" );
   else
      printf( "January 1, 2000 is a %s.\n",
              dname[date.tm_wday] );
}
```

It prints this:

```
January 1, 2000 is a Saturday.
```


modf

Library

ANSI

Syntax

```
#include <math.h>
double modf(double x, double *ip);
```

Description

modf() splits x into integral and fractional parts. It stores the integral part in *ip and returns the fractional part. Both parts carry the same sign as x.

Returns

The signed fractional part of x.

See Also

ceil(), floor(), frexp()

Example

This program prints the integral and fractional parts of any number you enter:

```
#include <stdio.h>
#include <math.h>

main()
{
   double x, integer, fraction;

   printf( "Enter a number: " );
   scanf( "%lf", &x );

   fraction = modf( x, &integer );

   printf( "Its integral part is %G\n"
           "Its fractional part is %G\n",
           integer, fraction );
}
```

This is sample output from the program:

```
Enter a number: 3.141593
Its integral part is 3
Its fractional part is 0.141593
```

open

Not in the ANSI standard.

Library

unix

Syntax

```
#include <unix.h>
#include <fcntl.h>
int open(char *filename, int mode);
```

Description

To learn how the standard libraries handle streams, see "Using Files and Streams" on page 15.

To learn how to specify file names, see "Naming files" on page 18

open() opens a filename for reading or writing, according to mode. It returns the file number of the open file.

mode determines how open() will open the file. To set it, OR together the appropriate flags, from those described below. They're defined in fcntl.h.

Opening Flags

mode must have exactly one flag from this group:

Flag	Description
O_RDONLY	Open this file for reading only.
O_WRONLY	Open this file for writing only.
O_RDWR	Open this file for reading and writing.

Creation Flags

mode may have any number of flags from this group:

Flag	Description
O_APPEND	Set the file position indicator to the end of the file prior to each write.
O_CREAT	Create the file if it doesn't already exist.
O_TRUNC	If the file exists, overwrite it.
O_EXCL	If the file exists, don't create it again, but set the file position indicator to the beginning of the existing file.

File-Type Flags

mode may have at most one flag from this group:

Flag	Description
O_TEXT	Text file.
O_BINARY	Binary file.

If neither file-type flag is set, open() uses the global variable _fmode, declared in fcntl.h. The initial value of _fmode is O_BINARY. Your program may change the value of _fmode anytime. The new value does not affect the mode of files opened with fopen().

If you open a file for updating (mode uses the flag O_RDWR), you can both read from and write to the file. However, you must call one of these functions when you want to switch between reading and writing: lseek(), fflush(), fseek(), fsetpos(), rewind(). In addition, you can start writing when a read operation reaches the end of the file.

See Also

close(), creat(), read(), write().

Example

This program opens a file named MyFile as a write-only text file that's overwritten if it already exits. The program writes a message to the file and closes it.

```
#include <unix.h>
#include <fcntl.h>

main()
{
   int fn;

   fn = open( "MyFile", O_WRONLY | O_CREAT |
                        O_TRUNC | O_TEXT );

   write( fn, "hello world\n", 12 );
   close( fn );
}
```

perror

Library ANSI, ANSI-small

Syntax

```
#include <stdio.h>
void perror(const char *s);
```

Description perror() prints an error message appropriate for the error id in errno.

perror() writes this to stderr:

userString: Error *id*

where *userString* is the argument s and *id* is the value of errno.

See Also assert(), strerror()

Example This program tries to compute the square root of a negative number and prints an error message:

```
#include <stdio.h>
#include <math.h>

main()
{
   double i, result;

   i = -4.0;
   result = sqrt(i);

   if ( (result==0) & (errno!=0) )
      perror( "Invalid argument for sqrt()" );
   else
      printf( "The square root of %f is %f\n",
              i, result );
}
```

The program prints this line:

```
Invalid argument for sqrt(): Error 33
```


pow

Library ANSI

Syntax

```
#include <math.h>
double pow(double x, double y);
```

Description pow() returns x^y, x raised to the power of y.

Returns pow() returns these values:

If...	The function returns...
x is greater than zero	x^y.
x is zero, y is less than or equal to zero	Negative Infinity. It also sets errno to EDOM.
x is negative, y is not an integer	Negative Infinity. It also sets errno to EDOM.
the result is too large	Infinity, with the same sign as x.

See Also exp(), ldexp(), log(), log10(), sqrt()

Example This program prints x^y for any x and y you enter:

```
#include <stdio.h>
#include <math.h>

main()
{
   double x, y, result;

   printf( "Enter x: " );
   scanf( "%lf", &x );
   printf( "Enter y: " );
   scanf( "%lf", &y );

   result = pow( x, y );

   printf( "%G to the power of %G is %G\n",
           x, y, result );
}
```

This is sample output from the program:

```
Enter x: 3.5
Enter.y: 2.1
3.5 to the power of 2.1 is 13.8849
```

printf

Library

ANSI, ANSI-small

Syntax

```
#include <stdio.h>
int printf(const char *format, ...);
```

Description

To learn how the standard libraries handle streams, see "Using Files and Streams" on page 15.

For complete information on format specifiers, read Chapter 4, "Using printf and scanf."

`printf()` writes a formatted string to standard output (`stdout`). It evaluates the data arguments (shown as "`...`" in the prototype), converts them according to the string `format`, and writes the result to standard output.

The format string `format` contains ordinary characters and format specifiers. `printf()` prints out the ordinary characters directly to the stream. The format specifiers tell it how to print the data arguments. A format specifier begins with a `%` and ends with a conversion character. For example, the format specifier `%d` prints out a decimal integer.

If `printf()` comes to an invalid format specifier, it prints out the converted string up to the specifier, and returns successfully.

Note

If you open a stream in text mode, `printf()` converts Line Feed characters to Returns. For more information, see "Binary and text files" on page 20.

Returns

`printf()` returns these values:

If the function...	It returns...
succeeds	the number of characters printed.
fails	EOF.

See Also

`fprintf()`, `fputc()`, `fputs()`, `scanf()`, `sprintf()`, `vfprintf()`, `putc()`, `putchar()`, `puts()`, `vprintf()`, `vsprintf()`

Example

This program writes a line to standard output.

```
#include <stdio.h>

main()
{
   fprintf( fp, "pi = %7.5f", 3.141593 );
}
```

The program writes this line to standard output:

```
pi = 3.14159
```


putc

Library

ANSI, ANSI-small

Syntax

```
#include <stdio.h>
int putc(char c, FILE *stream);
```

Description

To learn how the standard libraries handle streams, see "Using Files and Streams" on page 15.

For more information on the append mode, see `fopen()` on page 171.

`putc()` prints the character `c` to the stream `stream`. `putc()` is like `fputc()` except that `putc()` is a macro, not a function.

`putc()` writes the character to your current position in the stream, and moves your current position ahead. If you opened the file in append mode, `putc()` appends the character to the end of the file.

Note

If you open a stream in text mode, `fputc()` converts Line Feed characters to Returns. For more information, see "Binary and text files" on page 20.

Since `putc()` is a macro, it may evaluate `stream` more than once. Make sure that `stream` is not an expression that may have side effects.

Returns

`putc()` returns these values:

If the function...	It returns...
succeeds	the integer value of the character `c`.
fails (such as a read-only file)	EOF.

See Also

`fgetc()`, `fputc()`, `fputs()`, `fwrite()`, `fprintf()`, `getc()`, `getchar()`, `putchar()`, `puts()`, `ungetc()`

Example

This program writes the string "Hello File." to MyFile, character by character.

```
#include <stdio.h>

main()
{
   FILE *fp;
   char *s = "Hello File.";
   int i;

   fp = fopen( "MyFile", "w" );

   for ( i=0; s[i]!='\0'; i++ )
      putc( s[i], fp );

   fclose( fp );
}
```


putchar

Library

ANSI, ANSI-small

Syntax

```
#include <stdio.h>
int putchar(int c);
```

Description

putchar() prints the character c to standard output. It's similar to fputc(), with stdout as the output stream.

Returns

putchar() returns these values:

If the function...	It returns...
succeeds	the integer value of the character c.
fails (such as a read-only file)	EOF.

To learn how the standard libraries handle streams, see "Using Files and Streams" on page 15.

See Also

fgetc(), fputc(), fputs(), fwrite(), getc(), getchar(), printf(), putc(), puts(), ungetc()

Example

This program writes message to standard output, one character at a time:

```
#include <stdio.h>

main()
{
   char *s = "Hello out there!\n";
   int i = 0;

   while ( s[i] != '\0' ) {
      putchar( s[i] );
      i++;
   }
}
```

puts

Library

ANSI, ANSI-small

Syntax

```
#include <stdio.h>
int puts(const char *s);
```

Description

To learn how the standard libraries handle streams, see "Using Files and Streams" on page 15.

puts() writes a string and a newline ('\n') to standard output.

puts() writes the characters from s to stdout. When it reaches the null character ('\0'), it writes a newline ('\n') and returns.

Returns

puts() returns these values:

If the function...	It returns...
succeeds	0 (zero).
fails	EOF.

See Also

fgets(), fputc(), fputs(), fwrite(), gets(), printf(), putc(), ungetc()

Example

This program writes several lines to standard output.

```
#include <stdio.h>

main()
{
   puts( "This is only a test." );
   puts( "If this were a real program," );
   puts( "it would do something useful." );
}
```

The program writes out these lines:

```
This is only a test.
If this were a real program,
it would do something useful.
```


putw

Not in the ANSI standard.

Library

unix

Syntax

```
#include <unix.h>
int putw(int word, FILE *stream);
```

Description

To learn how the standard libraries handle streams, see "Using Files and Streams" on page 15.

putw() writes word as two bytes—the high byte and then the low byte—to the file stream.

putw() assumes the bytes are in memory order. On the MC68000 the most significant byte is in the first byte.

Returns

putw() returns these values:

If the function...	It returns...
succeeds	word
fails	EOF

See Also

getw(), write(), fwrite()

Example

This program writes the two bytes 'hi' to the file MyFile:

```
#include <unix.h>
#include <fcntl.h>

main()
{
   FILE *fp;

   fp = fopen( "MyFile", "w" );

   putw( 'hi', fp );

   fclose( fp );
}
```

qsort

Library

ANSI

Syntax

```
#include <stdlib.h>
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

Description

`qsort()` sorts an array in ascending order. It uses the quicksort algorithm.

The `compar` function compares the elements from the array. It should take two arguments, pointers to two array elements. Declare it like this: (You can use different names for the function and arguments.)

```
int compar( const void *e1, const void *e2 );
```

It returns an integer describing how the elements compare:

If the first element is...	compar returns...
less than the second	a negative integer.
equal to the second	0 (zero).
greater than the second	a positive integer.

These are the arguments to `qsort()`:

Argument	Description
`base`	Pointer to the array.
`nmemb`	Number of elements in the array.
`size`	Size of each element in the array.
`compar`	The comparison function, which takes two arguments.

Note

Make sure your `compar` function does not call `qsort()`. THINK C's implementation is not reentrant.

See Also

`bsearch()`

Example

This program creates an array of ten random integers and sorts it:

```
#include <stdlib.h>
#include <stdio.h>

#define ASIZE 10
#define AMAX 100

int compare( const void *n1, const void *n2 )
{
   return ( *((int *) n1) - *((int *) n2) );
}

main()
{
   int a[ASIZE], i;

   /* Initialize the random number generator. */
   srand( (unsigned int) clock() );

   /* Initialize the array.*/
   for ( i=0; i<ASIZE; i++ )
      a[i] = rand() % AMAX;

   printf( "Before qsort(): " );
   for ( i=0; i<ASIZE; i++ )
      printf( "%2d ", a[i] );
   putchar('\n');

   qsort( a, ASIZE, sizeof(int), compare );

   printf( "After qsort():  " );
   for ( i=0; i<ASIZE; i++ )
      printf( "%2d ", a[i] );
   putchar('\n');
}
```

This is sample output from the program:

```
Before qsort(): 35 22 67 79  1 31 11 42 33 54
After qsort():   1 11 22 31 33 35 42 54 67 79
```

raise

Library

ANSI, ANSI-small

Syntax

```
#include <signal.h>
int raise(int sig);
```

Description

`raise()` sends the signal `sig` to your program. If you specified a response to the signal, `raise()` performs it and sets the response back to the default. If you specify a function as the response, `raise()` calls it. If you didn't specify a response, `raise()` performs the default response. In THINK C, the default response is to end your program.

To specify a response to a signal, use `signal()`.

`sig` must be one of the following:

Value	Description
SIGABRT	Abnormal termination
SIGFPE	An erroneous arithmetic operation, like a zero divide.
SIGILL	An invalid function image, like an illegal instruction.
SIGINT	Interactive attention, like an interrupt.
SIGSEGV	Invalid access to storage, like an access outside memory limits
SIGTERM	A request to terminate the program.

The THINK C ANSI library raises two signals. The other signals are raised only if your program raises them.

It raises...	When you...
SIGABRT	call `abort()`, described on page 101.
SIGINT	press Command-. (Command-Period) in the console package interface.

Returns

`raise()` returns these values:

If the function...	It returns...
succeeds	0 (zero).
fails	Non-zero.

`raise()` might also set `errno` to this value:

It sets errno to...	If...
EINVAL	`sig` is not one of the correct values.

See Also

`signal()`, `kill()`

Example

This program raise the signal `SIGTERM` if the error number for `HandleError()` is 0:

```
#include <stdio.h>
#include <signal.h>

main()
{
   HandleError(0);
}

int HandleError( int err )
{
   char s[5];

   if ( err != 0 )
      printf( "Error #%d happened.\n", err );
   else {
      printf( "Fatal error #%d. "
              "Program terminated.\n", err );
      printf( "Press Return to continue.\n" );
      gets( s );
      raise( SIGTERM );
   }
}
```

rand

Library

ANSI

Syntax

```
#include <stdlib.h>
int rand(void);
```

Description

The `Random()` *function in the Macintosh Toolbox also generates random numbers. For more information, see Inside Macintosh I, Chapter 6, "QuickDraw."*

`rand()` returns a random integer in the range 0 to RAND_MAX. In THINK C, RAND_MAX is 32 767.

`rand()` generates its numbers in a sequence based on a seed. Initially, the seed is 1. To set it to a different number, use `srand()`. If you want `rand()` to return the same sequence of numbers as in a previous series of calls, set the seed to the same value.

Returns

A random integer in the range 0 to RAND_MAX.

See Also

`srand()`

Example

This program prints the same sequence of ten random numbers twice.

```
#include <stdio.h>
#include <stdlib.h>

main()
{
   int i, s;

   /* Pick a seed. */
   s = (unsigned int) clock();

   srand(s);
   for ( i=0; i<10; i++ )
      printf( "%5d\n", rand() );

   putchar('\n');

   srand(s);
   for ( i=0; i<10; i++ )
      printf( "%5d\n", rand() );
}
```


read

Not in the ANSI standard.

Library

`unix`

Syntax

```
#include <unix.h>
int read(int fn, char *buffer, unsigned int count);
```

Description

To learn how the standard libraries handle streams, see "Using Files and Streams" on page 15.

`read()` reads a number of bytes from a file into an buffer.

These are the arguments to `read()`:

Argument	Description
`fn`	File number of the file to read from
`buffer`	Buffer to read into
`count`	The number of bytes to read

`read( fn, buffer, sizeof(Foo)*3 )`, where `buffer` is declared as `Foo buffer[3]`, is identical to

```
fread( buffer, sizeof(Foo), 3, fp );
```

except that `read()` takes a file number (`fn`) instead of a file descriptor (`fp`).

Returns

`read()` returns these values:

If the function...	It returns...
succeeds	the number of bytes read
reaches the end of the file	the number of bytes read before the end of the file.
fails	EOF

See Also

`getw()`, `write()`, `fread()`

Example

This code fragment writes out 3 items of the structure `Foo` to the file `MyFooFile`, and then reads them back in.

```
#include <stdio.h>
#include <fcntl.h>
#include <unix.h>

typedef struct {
   int a;
   char b;
} Foo;
```

```
main ()
{
   int fn;
   int i;
   Foo inbuf[3];
   Foo outbuf[3] = { { 1, 'a' },
                     { 2, 'b' },
                     { 3, 'c' } };

   fn = open( "MyFooFile", O_WRONLY | O_CREAT |
                           O_TRUNC | O_BINARY );
   write( fn, (char *) &outbuf, sizeof(Foo)*3);
   lseek( fn, 0, SEEK_SET );

   read( fn, (char *) &inbuf, sizeof(Foo)*3 );

   for ( i=0; i<3; i++ )
      printf( "foo[%d] = { %d %c }\n",
              i, inbuf[i].a, inbuf[i].b );

   close( fn );
}
```

The program writes these lines:

```
foo[0] = { 1 a }
foo[1] = { 2 b }
foo[2] = { 3 c }
```


realloc

Library

ANSI, ANSI-small

Syntax

```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

Description

To learn how THINK C allocates memory, see "Allocating Memory" on page 20.

`realloc()` changes the size of a block of memory while preserving its contents. If necessary, `realloc()` copies the contents to another block of memory.

`ptr` must be a pointer to a block of memory that `calloc()`, `malloc()`, or `realloc()` returned. `size` is the desired new size for the block of memory.

If `ptr` is NULL, `realloc()` behaves like a call to `malloc()` for the specified size.

If the new size is smaller than the old size, `realloc()` takes away space at the end of the old block and discards the contents of that space.

If `size` is 0 (zero) and `ptr` is not NULL, `realloc()` behaves like a call to `free()` and frees the block that `ptr` points to.

If the new size cannot be allocated, `realloc()` leaves the region unchanged and returns NULL.

Returns

`realloc()` returns these values:

If the function...	It returns...
allocates the memory	A pointer to the new block of memory
can't allocate the memory	NULL.

See Also

`calloc()`, `free()`, `malloc()`

Example

This code fragment reads structures into a buffer, reallocating a larger buffer as necessary.

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
   int a, b;
} Foo, *FooPtr;

main()
{
   FooPtr buf = NULL;
   int curElement = 0;
   int size = 0;
   FILE *fp;

   fp = fopen( "foofile", "rb" );

   while (!feof(fp)) {
      if (curElement == size) {
         size += 100;
         buf = realloc( buf, size*sizeof(Foo) );
         if (buf == NULL)
            exit(0);
      }
      .
      .
      .
   }
}
```


remove

Library

ANSI, ANSI-small

Syntax

```
#include <stdio.h>
int remove(const char *filename);
```

Description

To learn how to specify file names, see "Naming files" on page 18.

`remove()` deletes the file named `filename`.

If the file is open, `remove()` doesn't delete it and returns unsuccessfully.

Returns

`remove()` returns these values:

If the function...	It returns...
succeeds	0 (zero).
fails	Non-zero.

See Also

`tmpfile()`

Example

This program deletes the file `MyFile`:

```
#include <stdio.h>

main()
{
   remove( "MyFile" );
}
```

rename

Library

ANSI, ANSI-small

Syntax

```
#include <stdio.h>
int rename(const char *old, const char *new);
```

Description

To learn how to specify file names, see "Naming files" on page 18

rename() changes the name of a file from old to new.

If there's already a file named new, rename() returns unsuccessfully, without changing the name of the file old

You can't use rename() to move a file to a different folder. Both old and name must be names for files in the same folder. For example, this statement would return unsuccessfully:

```
rename( "MyFile", "::OurFile" );
```

Returns

rename() returns these values:

If the function...	It returns...
succeeds	0 (zero).
fails	Non-zero.

Example

This program changes the name of MyFile to OurFile:

```
#include <stdio.h>

main()
{
    rename( "MyFile", "OurFile" );
}
```


rewind

Library

ANSI, ANSI-small

Syntax

```
#include <stdio.h>
void rewind(FILE *stream);
```

Description

For more information on files opened for updating, see fopen() on page 171.

rewind() sets the current file position for the file stream to the beginning of the file. It also clears the error indicator for the stream.

If you open a file for updating, you can call rewind() to switch between reading and writing to the file.

See Also

fseek(), fsetpos(), lseek()

Example

To learn how the standard libraries handle streams, see "Using Files and Streams" on page 15.

This program writes a string to a file, rewinds the file, and reads the string back in.

```
#include <stdio.h>

main()
{
    FILE *fp;
    char s[128];

    fp = fopen( "MyFile", "w+" );
    fputs( "This is only a test", fp );

    rewind( fp );

    fgets( s, 128, fp );
    printf( "String read: \"%s\"\n", s );
    fclose( fp );
}
```

The program prints this line:

```
String read: "This is only a test"
```

scanf

Library

ANSI

Syntax

```
#include <stdio.h>
int scanf(const char *format, ...);
```

Description

To learn how the standard libraries handle streams, see "Using Files and Streams" on page 15.

`scanf()` converts input from standard input (`stdin`) into a series of values, using a format string as its guide. Reading from the stream `stream`, it breaks up the input into a series of values according to `format`. It stores these values in the variables that the data arguments point to (shown as `...` in the prototype).

For complete information on format specifiers, read Chapter 4, "Using printf and scanf."

The format string `format` contains format specifiers, that tell `scanf()` how to convert the input. A format specifier begins with a `%` and ends with a conversion character. For example,%d converts some of the stream's contents to an integer. Characters that aren't in a format specifier must exactly match characters from the stream. `fscanf()` doesn't assign them to any data arguments.

`scanf()` returns unsuccessfully only it can't read from standard input. It returns successfully even if it doesn't assign values to all its data arguments.Since it returns the number of data arguments it assigned values to, you can tell whether it assigned to all of them.

Returns

`scanf()` returns these values:

If the function...	It returns...
converts any input	the number of data arguments it assigned values to.
converts no input	EOF.

See Also

`fgetc()`, `fgets()`, `fscanf()`, `getc()`, `getchar()`, `gets()`, `printf()`, `sscanf()`

Example

This program reads some values from standard input.

```
#include <stdio.h>

main()
{
   int x;

   printf( "Enter a number: " );

   scanf( "%d", &x );

   printf( "You entered %d\n", x );
}
```

setbuf

Library

ANSI, ANSI-small

Syntax

```
#include <stdio.h>
void setbuf(FILE *stream, char *buf);
```

Description

To learn how the standard libraries handle streams, see "Using Files and Streams" on page 15.

`setbuf()` lets you specify how to buffer the stream `stream`. You can set a stream's buffer or make the stream unbuffered. In THINK C, all streams are fully-buffered with 512-character buffers when they're opened.

If a stream is unbuffered, I/O functions read or write characters directly to the stream. If a stream is fully-buffered, I/O functions read (or write) a block of characters at once from (or to) a buffer. When you read from a stream, the function gives you characters from the buffer and fills it if it's empty. When you write to the stream, the function puts characters into the buffer and writes it to the file when it's full.

To make a stream unbuffered, call `setbuf()` with `buf` as NULL. To set a stream's buffer, call `setbuf()` with `buf` as the new buffer. The new buffer must be `BUFSIZ` (512) bytes big. (Since THINK C gives every stream a buffer of this size, you probably won't need to use `setbuf()` for this reason.)

To change how a console is buffered, use `csetmode()` on page 140.

Note

You can't change a stream's buffer after you read from or write to it. Make sure you call `setbuf()` after you open the stream, and before you read from or write to it.

See Also

`fflush()`, `setvbuf()`, `csetmode()`

Example

This program opens a file and prints its contents twice. The first time, the file is unbuffered. The second time, the file is fully-buffered. Notice how much quicker the file is written the second time around.

```
#include <stdio.h>

main()
{
   FILE *fp;
   int c;
   char *buf;

   fp = fopen( "MyFile", "r" );

   setbuf( fp, NULL );

   printf( "UNBUFFERED\n"
           "==========\n\n" );
   c = fgetc( fp );
   while ( c != EOF ) {
      putchar( c );
      c = fgetc( fp );
   }

   fclose( fp );
   fp = fopen( "MyFile", "r" );
   buf = (char *) malloc( BUFSIZ );

   setbuf( fp, buf );

   printf( "FULLY-BUFFERED\n"
           "==============\n\n" );
   c = fgetc( fp );
   while ( c != EOF ) {
      putchar( c );
      c = fgetc( fp );
   }

   fclose( fp );
   free( buf );
}
```

setjmp

Library

ANSI, ANSI-small

Syntax

```
#include <setjmp.h>
int setjmp(jmp_buf env);
```

Description

`setjmp()`, together with `longjmp()`, lets you jump back to a function earlier in the subroutine call chain.

`setjmp()` saves the current program environment in `env` and returns 0 (zero). Later, you can call `longjmp()` with `env` and `val` as arguments. Your program restores the environment `env` and continues running as if the `setjmp()` call just finished and returned the value `val`.

Don't use `longjmp()` if you haven't called `setjmp()` or if you called `setjmp()` in a function that's returned.

After you call `longjmp()`, some local variables may not have the right values. Those are variables that are local to the function that called `setjmp()` and changed value between the calls to `setjmp()` and `longjmp()`. To make sure that these variables have the correct values, declare them `volatile`. In this example below, `i` must be declared `volatile` to ensure that it has the right value.

Note

You can't use `setjmp()` and `longjmp()` in object methods.

See Also

`longjmp()`

Returns

`setjmp()` returns these values:

If it returns after...	It returns...
a direct call, to save the environment	0 (zero)
a call to `longjmp()`, to restore the environment	`val`. If `val` is 0, it returns 1.

Example

This program shows you how to use `setjmp()` and `longjmp()`. Note that `i` must be declared `volatile` to have the right value.

```
#include <stdio.h>
#include <setjmp.h>

void foo( void );
jmp_buf env;

main()
{
   volatile int i = 9;

   if ( setjmp(env) == 0 )
      printf( "Called setjmp() to set up a "
              "return point. i = %d\n", i);
   else {
      printf( "Returned from longjmp(). "
              "i = %d\n", i);
      exit();
   }

   i = 10;
   foo();
}

void foo( void )
{
   printf( "Now in foo().\n");

   longjmp( env, 1 );

   printf( "This line won't be printed.\n" );
}
```

This is what the program prints:

```
Called setjmp() to set up a return point. i = 9
Now in foo().
Returned from longjmp(). i = 10
```

setlocale

Library

ANSI

Syntax

```
#include<locale.h>
char *setlocale(int category, const char *locale);
```

Description

`setlocale()` is provided for ANSI C compatibility only. THINK C supports only the minimal "C" locale environment. It doesn't change your program's locale. It returns "C" if locale is either "C" or NULL. Otherwise, it returns NULL. It ignores `category`.

According to the standard, `setlocale()` changes part or all of your program's locale. These macros are for the `category` argument. They specify how to modify the program's locale.

Macro	Description
LC_ALL	Use the locale for all categories.
LC_COLLATE	Change the locale for the `strcoll()` and `strxfrm()` functions.
LC_TYPE	Change the locale for the character-handling functions and the multi-byte functions.
LC_MONETARY	Change the locale for monetary formatting.
LC_NUMERIC	Change the locale for numeric formatting.
LC_TIME	Change the locale for the `strftime()` function.

To localize your software for other countries, use the International Utilities Package, described in *Inside Macintosh VI*, Chapter 14, "Worldwide Software Overview."

Returns

`setlocale()` returns these values:

If locale is...	The function returns...
"C" or NULL	"C".
anything else	NULL.

setvbuf

Library

ANSI, ANSI-small

Syntax

```
#include <stdio.h>
int setvbuf(FILE *stream, char *buf, int mode,
            size_t size);
```

Description

To learn how the standard libraries handle streams, see "Using Files and Streams" on page 15.

`setvbuf()` lets you specify how to buffer the stream `stream`. You can set the size of a stream's buffer or make the stream unbuffered. In THINK C, all streams are fully-buffered with 512-character buffers when they're opened.

If a stream is unbuffered, I/O functions read or write characters directly to the stream. If a stream is fully-buffered, I/O functions read or write a block of characters at once to a buffer. When you read from a stream, the function gives you characters from the buffer and fills it if it's empty. When you write to the stream, the function puts characters into the buffer and writes it to the file when it's full.

To set the size of a stream's buffer, call `setvbuf()` with `buf` as the new buffer or NULL, mode as `_IOFBF`, and `size` as the new size. `size` should be a multiple of 512. If `buf` is NULL, `setvbuf()` allocates a buffer of size `size`. If `buf` is a buffer you supply, `setvbuf()` sets the stream's buffer to `buf`. `size` must be the size of the buffer.

To make a stream unbuffered, call `setvbuf()` with mode set to `_IONBF`. It ignores the values of `buf` and `size`.

To change how a console is buffered, use `csetmode()` on page 140.

Note

You can't change a stream's buffer after you read from or write to it. Make sure you call `setvbuf()` after you open the stream, and before you read from or write to it.

This table describes of the possible values for `mode`:

If mode is...	The file is buffered this way...
_IOFBF	Fully buffered.
_IOLBF	Line buffered. In THINK C, line-buffered streams are treated like fully-buffered streams.
_IONBF	Unbuffered.

Returns

`setvbuf()` returns these values:

If the function...	It returns...
succeeds	0 (zero)
fails	Non-zero.

See Also

`fflush()`, `setbuf()`, `csetmode()`

Example

This program opens a file and prints its contents twice. The first time, the file is unbuffered. The second time, the file is fully-buffered. Notice how much quicker the file is written the second time around.

```
#include <stdio.h>

main()
{
   FILE *fp;
   int c;

   fp = fopen( "MyFile", "r" );

   setvbuf( fp, NULL, _IONBF, 0 );

   printf( "UNBUFFERED\n"
           "==========\n\n" );
   c = fgetc( fp );
   while ( c != EOF ) {
      putchar( c );
      c = fgetc( fp );
   }

   fclose( fp );
   fp = fopen( "MyFile", "r" );

   setvbuf( fp, NULL, _IOFBF, 1024 );

   printf( "FULLY-BUFFERED\n"
           "==============\n\n" );
   c = fgetc( fp );
   while ( c != EOF ) {
      putchar( c );
      c = fgetc( fp );
   }

   fclose( fp );
}
```


signal

Library

ANSI, ANSI-small

Syntax

```
#include <signal.h>
void (*signal(int sig, void (*func)(int)))(int);
```

Description

`signal()` tells your program how to respond to the signal `sig`. You can ignore the signal, exit your program, or call a function. `signal()` returns a pointer to another function that returns an `int`.

To raise a signal, call `raise()`.

`func` specifies how your program should respond to `sig`. It can be one of the following:

If func is...	Your program does this when sig is raised...
`SIG_IGN`	Ignores the signal.
`SIG_DFL`	Performs the default response, which is to exit the program abnormally.
a function	Calls the function, with `sig` as its argument.

Unless you set a response for a signal with `signal()`, `raise()` performs the response for `SIG_DFL`.

If `func` is a pointer to a function, declare it like this: (You can use different names for the function and argument.)

```
void *func( int sig );
```

Before `raise()` calls your function, it sets the response to `SIG_DFL`.

`sig` must be one of the following signals:

Value	Description
`SIGABRT`	Abnormal termination.
`SIGINT`	Interactive attention, like an interrupt.
`SIGFPE`	An erroneous arithmetic operation, like a zero divide.
`SIGILL`	An invalid function image, like an illegal instruction.
`SIGSEGV`	Invalid access to storage, like an access outside memory limits
`SIGTERM`	A request to terminate the program.

The THINK C ANSI library raises two signals. The other signals are raised only if your program raises them.

It raises...	When you...
SIGABRT	call abort(), described on page 101.
SIGINT	press Command-. (Command-Period) in the console package interface.

Returns

signal() returns these values:

If the function...	It returns...
succeeds	The previous response for the signal sig.
fails	SIG_ERR. It also sets errno to a positive value.

signal() might also set errno to this value:

It sets errno to...	If...
EINVAL	sig is not one of the correct values.

See Also

raise(), kill()

Example

This program sets up a response for SIGINT and then prints the file MyFile to a console window. When you press Command-. (Command-Period), raise() calls OKExit(), which asks you if you really want to exit the program:

```
#include <stdio.h>
#include <console.h>
#include <signal.h>

/* To be portable, define the flag that */
/* the signal handler sets like this.   */
volatile sig_atomic_t sigFlag;

/* The signal handler should be a short   */
/* function that resets the signal and    */
/* sets a global flag.                    */
void sigHandler( int sig )
{
   signal( SIGINT, sigHandler );
   sigFlag = 1;
}
```



```
/* This function makes sure   */
/* the user wants to exit.    */
int OKExit()
{
   char c;

   do {
      printf( "\a\n\n" );
      printf( "Do you want to exit? (y or n) ");
      c = getchar();
   } while ( (c!='y') && (c!='n') );

   printf( "\n\n" );
   if (c=='y')
      return 1;
   else {
      sigFlag = 0;
      return 0;
   }
}

main()
{
   char c;
   FILE *fp;

   /* Set up the console window. */
   cshow(stdout);
   csetmode(C_CBREAK, stdout);
   console_options.pause_atexit = 0;

   /* Set up the SIGINT response. */
   signal( SIGINT, sigHandler );
   sigFlag = 0;

   /* Open MyFile and print it. */
   fp = fopen( "MyFile", "r" );
   for ( c=getc(fp); c!=EOF; c=getc(fp) ) {
      putchar( c );

      /* If the user pressed Command-Period  */
      /* and wants to exit, stop printing    */
      /* the file.                           */
      if (sigFlag && OKExit())
         break;
   }
   fclose( fp );
}
```

sin

Library ANSI

Syntax

```
#include <math.h>
double sin(double x);
```

Description `sin()` returns the sine of the argument x.

Returns sin x, in radians.

See Also `acos()`, `asin()`, `atan()`, `atan2()`, `cos()`, `sinh()`, `tan()`

Example This program prints the sine of any number you enter:

```
#include <stdio.h>
#include <math.h>

main()
{
   double x, result;

   printf( "Enter an angle: " );
   scanf( "%lf", &x );

   result = sin(x);

   printf( "sin(%G) = %G\n", x, result);
}
```

This is sample output from the program:

```
Enter an angle: 13.5
sin(13.5) = 0.803784
```


sinh

Library ANSI

Syntax

```
#include <math.h>
double sinh(double x);
```

Description `sinh()` returns the hyperbolic sine of the argument x.

Returns `sinh()` returns these values:

If the function...	It returns...
succeeds	sinh x.
fails since x is greater than HUGE_VAL	Infinity. It also sets `errno` to ERANGE.

See Also `asin()`, `cosh()`, `sin()`, `tanh()`

Example This program prints the hyperbolic sine of any number you enter:

```
#include <stdio.h>
#include <errno.h>
#include <math.h>

main()
{
   double x, result;

   printf( "Enter an angle: " );
   scanf( "%lf", &x );

   result = sinh(x);

   printf( "sinh(%G) = %G\n", x, result);
}
```

This is sample output from the program:

```
Enter an angle: 3.0
sinh(3) = 10.0179
```

sleep

Not in the ANSI standard.

Library

unix

Syntax

```
#include <unix.h>
void sleep(unsigned int secs);
```

Description

`sleep()` waits `secs` seconds and then continues.

Example

This program prints "hello," waits five seconds, and prints "good-bye."

```
#include <unix.h>
#include <stdio.h>

main()
{
   printf( "hello\n" );

   sleep( 5 );

   printf( "good-bye\n" );
}
```


sprintf

Library

ANSI, ANSI-small, ANSI-A4

Syntax

```
#include <stdio.h>
int sprintf(char *s, const char *format, ...);
```

Description

`sprintf()` writes a formatted output to a string. It evaluates the data arguments (shown as "..." in the prototype), converts them according to the string `format`, and writes the result to the string `s`, terminated with a null character (`'\0'`).

For complete information on format specifiers, read Chapter 4, "Using printf and scanf."

The format string `format` contains ordinary characters and format specifiers. `sprintf()` transfers the ordinary characters directly to the string `s`. The format specifiers tell it how to write the data arguments. A format specifier begins with a % and ends with a conversion character. For example, the format specifier %d writes a decimal integer.

If `sprintf()` comes to an invalid format specifier, it sets `s` to the converted string up to the specifier, and returns successfully.

Returns

`sprintf()` returns these values:

If the function...	It returns...
succeeds	the number of characters printed.
fails	EOF.

See Also

`fprintf()`, `printf()`, `sscanf()`, `vfprintf()`, `vprintf()`, `vsprintf()`

Example

This program formats a line into a string and prints the string to standard output.

```
#include <stdio.h>

main()
{
   char s[80];

   sprintf( s, "pi = %7.5f", 3.141593 );

   puts( s );
}
```

The program writes this line:

```
pi = 3.14159
```

sqrt

Library ANSI

Syntax

```
#include <math.h>
double sqrt(double x);
```

Description `sqrt()` returns the square root of x.

Returns `sqrt()` returns these values:

If x is...	The function returns
not negative	$\sqrt{x}$.
negative	0 (zero). It also sets `errno` to EDOM.

See Also `pow()`

Example This program prints the square root of any number you enter:

```
#include <stdio.h>
#include <math.h>

main()
{
   double x, result;

   printf( "Enter a number: " );
   scanf( "%lf", &x );

   result = sqrt(x);

   printf( "sqrt(%G) = %G\n", x, result);
}
```

This is sample output from the program:

```
Enter a number: 2389.3
sqrt(2389.3) = 48.8805
```


srand

Library ANSI

Syntax

```
#include <stdlib.h>
void srand(unsigned int seed);
```

Description `srand()` initializes the random number generator, using its argument as the seed for a new sequence. Use `rand()` to produce a random number.

If you don't call `srand()`, the seed is 1. If you want `rand()` to return the same sequence of numbers as in a previous series of calls, set the seed to the same value.

See Also `rand()`

Example This program prints the same sequence of ten random numbers twice.

The `Random()` function in the Macintosh Toolbox also generates random numbers. For more information, see Inside Macintosh I, *Chapter 6, "QuickDraw."*

```
#include <stdio.h>
#include <stdlib.h>

main()
{
   int i, s;

   /* Pick a seed. */
   s = (unsigned int) clock();

   srand(s);
   for ( i=0; i<10; i++ )
      printf( "%5d\n", rand() );

   putchar('\n');

   srand(s);
   for ( i=0; i<10; i++ )
      printf( "%5d\n", rand() );
}
```

sscanf

Library

ANSI, ANSI-small, ANSI-A4

Syntax

```
#include <stdio.h>
int sscanf(const char *s, const char *format, ...)
```

Description

`sscanf()` converts a string's contents into a series of values, using a format string as its guide. Reading from the string `s`, it breaks up the input into a series of values according to `format`. It stores these values in the variables that the data arguments point to (shown as ". . ." in the prototype).

The format string `format` contains format specifiers, that tell `sscanf()` how to convert the string's contents. A format specifier begins with a % and ends with a conversion character. For example, %d converts some of the string's contents to an integer. Characters that aren't in a format specifier must exactly match characters from the string `s`. `sscanf()` doesn't assign them to any data arguments.

For complete information on format specifiers, read Chapter 4, "Using printf and scanf."

`sscanf()` returns unsuccessfully only it can't read from `s`. It returns successfully even if it doesn't assign values to all its data arguments.Since it returns the number of data arguments it assigned values to, you can tell whether it assigned to all of them.

Returns

`sscanf()` returns these values:

If the function...	It returns...
converts any input	the number of data arguments it assigned values to.
converts no input	EOF.

See Also

`atof()`, `atoi()`, `atol()`, `fscanf()`, `scanf()`, `sprintf()`, `strtod()`, `strtol()`, `strtoul()`,

Example

This program converts the number from a string to a floating-point value:

```
#include <stdio.h>

main()
{
   char *s = "3.141593";
   float pi;

   sscanf( s, "%f", &pi );
   printf( "pi = %f\n", pi );
}
```


strcat

Library

ANSI, ANSI-small, ANSI-A4

Syntax

```
#include <string.h>
char *strcat(char *s1, const char *s2)
```

Description

`strcat()` appends two strings. It appends a copy of the string `s2` to the end of the string `s1`, overwriting the null character (`'\0'`) at the end of `s1`.

Make sure there is enough space for string `s2` in the character array after the end of the string `s1`.

To copy a block of memory that's not null-terminated, use `memcpy()` or `memmove()`.

Returns

The value of `s1`, with `s2` appended.

See Also

`memcpy()`, `memmove()`, `strcpy()`, `strncat()`, `strncpy()`

Example

This program appends two strings and writes the result.

```
#include <stdio.h>
#include <string.h>

main()
{
   char s1[25] = "hello ";
   char *s2 = "world";

   printf( "Before strcat(): "
           "s1 = \"%s\"\n", s1 );

   strcat( s1, s2 );

   printf( "After strcat():  "
           "s1 = \"%s\"\n", s1 );
}
```

The program prints these lines:

```
Before strcat(): s1 = "hello "
After strcat():  s1 = "hello world"
```

strchr

Library ANSI, ANSI-small, ANSI-A4

Syntax

```
#include <string.h>
char *strchr(const char *s, int c);
```

Description `strchr()` looks for the character `c` in the string `s`. It returns a pointer to first occurrence of `c` in `s`. If `c` is the null character (`'\0'`), `strrchr()` returns a pointer to the null character terminating `s`.

Make sure that `c` is between 0 and 255. `strchr()` converts it to an `unsigned char`.

To search for a character in a block of memory that's not null-terminated, try `memchr()`.

Returns `strchr()` returns these values:

If the function...	It returns...
finds `c` in `s`	a pointer to the first occurrence of `c`.
can't find `c` in `s`	NULL.

See Also `memchr()`, `strcspn()`, `strpbrk()`, `strrchr()`, `strspn()`, `strstr()`

Example This program looks for the character `t` in `"Symantec"`:

```
#include <stdio.h>
#include <string.h>

main()
{
   char s[20] = "Symantec";
   char c = 't';

   printf( "s = \"%s\"\n", s );
   printf( "c = '%c'\n", c );
   printf( "strchr(s,c) = \"%s\"\n",
           strchr(s,c) );
}
```

It prints these lines:

```
s = "Symantec"
c = 't'
strchr(s,c) = "tec"
```


strcmp

Library ANSI, ANSI-small, ANSI-A4

Syntax

```
#include <string.h>
int strcmp(const char *s1, const char *s2);
```

Description `strcmp()` compares the string `s1` with the string `s2`. This function compares the numeric values of the characters in the string. It works with ASCII character set and the Macintosh extensions.

To compare two blocks of memory that aren't null-terminated, use `memcmp()`.

Returns `strcmp()` returns these values:

If s1 is...	The function returns...
greater than `s2`	a positive integer.
equal to `s2`	0 (zero)
less than `s2`	a negative integer.

See Also `memcmp()`, `strncmp()`

Example This program asks you to enter two strings and compares them:

```
#include <stdio.h>
#include <string.h>

main()
{
   char s1[80], s2[80];
   int result;

   printf( "Enter the first string: ");
   scanf( "%s", s1 );

   printf( "Enter the second string: ");
   scanf( "%s", s2 );

   result = strcmp( s1, s2 );

   if ( result > 0 )
     printf( "s1 is greater than s2.\n" );
   else if ( result < 0 )
     printf( "s1 is less than s2.\n" );
   else
     printf( "s1 and s2 are the same.\n" );
}
```

strcoll

Library

ANSI

Syntax

```
#include <string.h>
int strcoll(const char *s1, const char *s2);
```

Description

`strcoll()` compares two strings. It's equivalent to

```
strcmp( s1, s2 );
```

`strcoll()` is provided for ANSI C compatibility only. THINK C supports only the minimal locale environment. According to the standard, `strcoll()` compares the string `s1` to the string `s2`, interpreting both strings as appropriate to the current locale.

To localize your software for other countries, use the International Utilities Package, described in *Inside Macintosh VI*, Chapter 14, "Worldwide Software Overview."

See Also

`memcmp()`, `strcmp()`, `strncmp()`

Returns

`strcoll()` returns these values:

If s1 is...	The function returns...
greater than s2	a positive integer.
equal to s2	0 (zero)
less than s2	a negative integer.

strcpy

Library

ANSI, ANSI-small, ANSI-A4

Syntax

```
#include <string.h>
char *strcpy(char *s1, const char *s2);
```

Description

`strcpy()` copies the string `s2` into the array `s1`, including the terminating null character (`'\0'`).

To copy a block of memory that's not null-terminated, use `memcpy()` or `memmove()`.

Returns

`s1`, after copying the contents of `s2` into it.

See Also

`memcpy()`, `memmove()`, `strcat()`, `strncat()`, `strncpy()`

Example

This program copies the contents of one string into another:

```
#include <stdio.h>
#include <string.h>

main()
{
    char *source = "good-bye";
    char dest[25] = "hello there";

    printf( "Before strcpy(): "
            "dest = \"%s\"\n", dest );

    strcpy( dest, source );

    printf( "After strcpy():  "
            "dest = \"%s\"\n", dest );
}
```

The program prints these lines:

```
Before strcpy(): dest = "hello there"
After strcpy():  dest = "good-bye"
```

strcspn

Library ANSI, ANSI-small, ANSI-A4

Syntax

```
#include <string.h>
size_t strcspn(const char *s1, const char *s2);
```

Description `strcspn()` returns the length of the initial segment of `s1` that contains no characters from `s2`. That value is also the position of the first character in `s1` that is in the set of characters in `s2`.

Returns The length of the segment of `s1` that contains no characters from `s2`.

See Also `strchr()`, `strpbrk()`, `strrchr()`, `strspn()`, `strstr()`

Example This program asks you to enter a string and finds the first digit in it:

```
#include <stdio.h>
#include <string.h>

main()
{
   char *digits = "1234567890";
   char string[80];
   int pos;

   printf( "Enter a string: " );
   gets( string );

   pos = strcspn( string, digits );

   printf( "The first digit is '%c', at "
           "position %d.\n", string[pos], pos );
}
```

This is sample output from the program:

```
Enter a string: THINK C is the #1 language.
The first digit is '1', at position 16.
```


strerror

Library ANSI, ANSI-small

Syntax

```
#include <string.h>
char *strerror(int errnum);
```

Description `strerror()` returns an error message appropriate for the error id in `errnum`.

`strerror()` returns a string of this form:

```
Error id
```

where *id* is the value of `errnum`.

Returns A pointer to the error message string.

See Also `assert()`, `perror()`

Example This program tries to compute the square root of a negative number and prints an error message:

```
#include <stdio.h>
#include <errno.h>
#include <math.h>
#include <string.h>

main()
{
   double i, result;
   char *msg;

   i = -4.0;
   result = sqrt(i);

   if ( (result==0) & (errno!=0) ) {
      msg = strerror( errno );
      printf( "sqrt() failed: %s.\n", msg );
   }
   else
      printf( "The square root of %G is %G\n",
              i, result );
}
```

The program prints this line:

```
sqrt() failed: Error 33
```

strftime

Library

ANSI

Syntax

```
#include <time.h>
size_t strftime(char *s, size_t maxsize,
                const char *format,
                const struct tm *timeptr);
```

Description

`strftime()` formats a time. It writes to the string `s` the time in `timeptr`, using the string `format` as a guide. If the time (including a final null character (`'\0'`)) takes up more than `maxsize` characters, `strftime()` returns 0.

`format` contains specifiers and other charcters. `strftime()` replaces the specifiers with values from `timeptr`. `strftime()` copies the other characters directly into `s`.

These are the specifiers for `format`:

Spec.	Replaced with the:
`%a`	abbreviated weekday name.
`%A`	full weekday name.
`%b`	abbreviated month name.
`%B`	full month name.
`%c`	date and time in the form `Wed Sep 19 18:23:07 1990`
`%d`	day of the month as a decimal number (`01-31`).
`%H`	hour (24-hour clock), as decimal number (`00-23`).
`%I`	hour (12-hour clock), as a decimal number (`01-12`).
`%j`	day of the year, as a decimal number (`001-366`).
`%m`	month, as a decimal number (`01-12`).
`%M`	minute, as a decimal number (`00-59`).
`%p`	AM/PM designation for a 12-hour clock.
`%S`	second, as a decimal number (`00-61`).
`%U`	week number of the year (with Sunday as the first day of the first week), as a decimal number (`00-53`).
`%w`	weekday, as a decimal number (`0` (Sunday) – `6` (Saturday)).
`%W`	week number of the year (with Monday as the first day of the first week), as a decimal number (`00-53`).

Spec.	Replaced with:
`%x`	date in the form `9/19/90`.
`%X`	time in the form `6:23 PM`.
`%y`	year without century, as a decimal number (`00-99`).
`%Y`	year with century, as a decimal number.
`%Z`	THINK C ignores this specifier. According to the standard, it prints the time zone if it can be determined.
`%%`	%

Returns

`strftime()` returns these values:

If the string created is...	The function returns...
not longer than `maxsize`	the length of the string created.
longer than `maxsize`	0 (zero). The string is not returned in the argument `s`.

See Also

`asctime()`, `ctime()`, `localtime()`, `mktime()`, `time()`

Example

This program prints the current date and time:

```
#include <stdio.h>
#include <time.h>

main()
{
   time_t now;
   struct tm *date;
   char s[80];

   now = time( NULL );
   date = localtime( &now );

   strftime( s, 80, "%A, %B %d, %Y %I:%M:%S %p",
             date );

   printf( "%s\n", s );
}
```

The program writes the date and time like this:

```
Wednesday, September 19, 1990 06:23:17 PM
```

strlen

Library ANSI, ANSI-small, ANSI-A4

Syntax

```
#include <string.h>
size_t strlen(const char *s);
```

Description `strlen()` returns the length of the string `s`. It counts the number of characters in the string until it reaches the final null character (`'\0'`).

Returns The length of the string `s`.

Example This program asks you to enter a string and prints its length:

```
#include <stdio.h>
#include <string.h>

main()
{
   char s[80];
   size_t len;

   printf( "Enter a string: " );
   gets( s );

   len = strlen( s );

   printf( "Your string has %ld characters.\n",
           (long) len );
}
```

This is sample output from the program:

```
Enter a string: hello world.
Your string has 12 characters.
```


strncat

Library ANSI, ANSI-small, ANSI-A4

Syntax

```
#include <string.h>
char *strncat(char *s1, const char *s2, size_t n)
```

Description `strncat()` appends n characters of one string to another. It copies characters from `s2` to the end of `s1` until it copies n characters or reaches the end of string `s2`. The first character of `s2` overwrites the null character (`'\0'`) at the end of `s1`.

If `s2` is less than n characters long, `strncat()` appends all of `s2` to `s1`.

To copy a block of memory that's not null-terminated, use `memcpy()` or `memmove()`.

Returns `strncat()` returns these values:

If n is...	The function returns...
not 0 (zero)	s1 with s2 appended
0 (zero)	s1 unchanged.

See Also `memcpy()`, `memmove()`, `strcat()`, `strcpy()`, `strncpy()`

Example This program appends some strings together:

```
#include <stdio.h>
#include <string.h>

main()
{
   char s1[20] = "hello ";
   char *s2 = "world.";
   char s3[20] = "What's up ";
   char *s4 = "doctor";

   printf( "Before: s1 = \"%s\" s2 = \"%s\"\n",
           s1, s2 );
   strncat( s1, s2, 10);
   printf( "After:  s1 = \"%s\"\n\n", s1 );

   printf( "Before: s3 = \"%s\" s4 = \"%s\"\n",
           s3, s4 );
   strncat( s3, s4, 3 );
   printf( "After:  s3 = \"%s\"\n", s3 );
}
```

The program prints these lines:

```
Before: s1 = "hello " s2 = "world."
After:  s1 = "hello world."

Before: s3 = "What's up " s4 = "doctor"
After:  s3 = "What's up doc"
```


strncmp

Library

ANSI, ANSI-small, ANSI-A4

Syntax

```
#include <string.h>
int strncmp(const char *s1, const char *s2, size_t n);
```

Description

`strncmp()` compares the first n characters of `s1` to the first n characters of `s2`. If it reaches a null character (`'\0'`), it stops comparing the strings and returns that the longer one is greater.

To compare two blocks of memory that are not null-terminated, use `memcmp()`.

Returns

`strncmp()` returns these values:

If the first n characters of s1 are...	The function returns...
greater than those in s2	a positive integer.
equal to those in s2	0 (zero).
less than those in s2	a negative integer.

See Also

`memcmp()`, `strcmp()`

Example

This program asks you to enter two strings and compares the first three characters of each:

```
#include <stdio.h>
#include <string.h>

main()
{
   char s1[80], s2[80];
   int result;

   printf( "Enter a string: " );
   gets( s1 );

   printf( "Enter a string: " );
   gets( s2 );

   result = strncmp( s1, s2, 3 );
```

```
    if ( result > 0 )
       printf( "\"%.3s\" is greater than "
               "\"%.3s\"\n", s1, s2 );
    else if ( result < 0 )
       printf( "\"%.3s\" is less than "
               "\"%.3s\"\n", s1, s2 );
    else
       printf( "\"%.3s\" is equal to "
               "\"%.3s\"\n", s1, s2 );
}
```

This is output from the program:

```
Enter a string: Jefferson
Enter a string: Jeff
"Jef" is equal to "Jef"

Enter a string: a
Enter a string: aa
"a" is less than "aa"
```


strncpy

Library ANSI, ANSI-small, ANSI-A4

Syntax

```
#include <string.h>
char *strncpy(char *s1, const char *s2, size_t n);
```

Description `strncpy()` copies n characters from `s2` into `s1`, overwriting whatever was in `s1`. If the length of `s2` is less than n, `strncpy()` appends a null character (`'\0'`) to the end of `s1`.

To copy a block of memory that's not null-terminated, use `memcpy()` or `memmove()`.

Returns `s1`, after copying the first n characters of `s2` into it.

See Also `memcpy()`, `memmove()`, `strcat()`, `strcpy()`, `strncat()`

Example This program copies three characters of one string to another:

```
#include <stdio.h>
#include <string.h>

main()
{
   char *s1 = "jumpy";
   char *s2 = "happening";

   printf( "Before: s1 = \"%s\" s2= \"%s\"\n",
           s1, s2);

   strncpy( s1, s2, 3 );

   printf( "After:  s1 = \"%s\"\n", s1 );
}
```

This is what the program prints:

```
Before: s1 = "jumpy" s2= "happening"
After:  s1 = "happy"
```

strpbrk

Library

ANSI, ANSI-small, ANSI-A4

Syntax

```
#include <string.h>
char *strpbrk(const char *s1, const char *s2);
```

Description

`strpbrk()` finds the first character in a string that's also in a set of characters. It returns a pointer to the first character in `s1` that's also in `s2`.

Returns

`strpbrk()` returns these values:

If the function...	It returns...
finds a character from `s2` in `s1`	a pointer to the first character in `s1` that's in `s2`.
doesn't find a character from `s2` in `s1`	NULL.

See Also

`memchr()`, `strchr()`, `strcspn()`, `strpbrk()`, `strrchr()`, `strspn()`, `strstr()`

Returns

This program finds the first vowel in *street*.

```
#include <stdio.h>
#include <string.h>

main()
{
   char *s = "street";
   char *set = "aeiou";
   char *result;

   result = strpbrk( s, set );
   printf( "strpbrk(\"%s\",\"%s\") == \"%s\"\n",
           s, set, result );
}
```

The program prints this line:

```
strpbrk("street","aeiou") == "eet"
```


strrchr

Library

ANSI, ANSI-small, ANSI-A4

Syntax

```
#include <string.h>
char *strrchr(const char *s, int c);
```

Description

`strrchr()` finds the last occurrence of a character in a string. It returns a pointer to the last occurrence of `c` in the string `s`.

If `c` is the null character (`'\0'`), `strrchr()` returns a pointer to the null character terminating `s`.

Make sure that `c` is between 0 and 255. `strrchr()` converts it to an `unsigned char`.

To search for a character in a block of memory that's not null-terminated, use `memchr()`.

Returns

`strrchr()` returns these values:

If the function...	It returns
finds `c` in `s`	a pointer to the last occurrence of `c` in `s`.
doesn't find `c` in `s`	NULL.

See Also

`memchr()`, `strchr()`, `strcspn()`, `strpbrk()`, `strpbrk()`, `strspn()`, `strstr()`

Example

This program finds the last `s` in `"Mississippi"`.

```
#include <stdio.h>
#include <string.h>

main()
{
   char c = 's';
   char *s = "Mississippi";
   char *result;

   result = strrchr( s, c );
   printf( "strrchr(\"%s\",'%c') == \"%s\"\n",
           s, c ,result );
}
```

The program prints this line:

```
strrchr("Mississippi",'s') == "sippi"
```

strspn

Library ANSI, ANSI-small, ANSI-A4

Syntax

```
#include <string.h>
size_t strspn(const char *s1, const char *s2)
```

Description `strspn()` returns the length of the initial segment of `s1` that contains only characters from `s2`. That value is also the position of the first character in `s1` that is not in the set of characters in `s2`.

Returns The length of the segment of `s1` that contains only characters from `s2`.

See Also `memchr()`, `strchr()`, `strcspn()`, `strpbrk()`, `strrchr()`, `strstr()`

Example This example finds the first consonant in `"aesop"`.

```
#include <stdio.h>
#include <string.h>

main()
{
   char *s = "aesop";
   char *set = "aeiou";
   int pos;

   result = strspn( s, set );
   printf( "The first consonant in \"%s\" "
           "is '%c',\n"
           "at position %d.\n",
           s, s[pos], pos );
}
```

The program writes this:

```
The first consonant in "aesop" is 's',
at position 2.
```


strstr

Library ANSI, ANSI-small, ANSI-A4

Syntax

```
#include <string.h>
char *strstr(const char *s1, const char *s2);
```

Description `strstr()` looks for a substring within a string. It returns a pointer to the first occurrence of the string `s2` in the string `s1`.

`strstr()` does not try to match the null character ending `s2`.

Returns `strstr()` returns these values:

If s2 is...	The function returns...
in s1	a pointer to the first occurrence of it in s1.
not in s1	NULL.
the null string ("")	s1.

See Also `memchr()`, `strchr()`, `strcspn()`, `strpbrk()`, `strrchr()`, `strspn()`

Example This program looks for the letters `tt` in `"scatter"`.

```
#include <stdio.h>
#include <string.h>

main()
{
   char *string = "scatter";
   char *substring = "tt";
   char *result;

   result = strstr( string, substring );
   printf( "strstr(\"%s\",\"%s\") == \"%s\"\n",
           string, substring, result );
}
```

The program prints this:

```
strstr("scatter","tt") == "tter"
```

strtod

Library

ANSI

Syntax

```
#include <stdlib.h>
double strtod(const char *nptr, char **endptr);
```

Description

`strtod()` converts the string `nptr` to a double precision floating point value. It skips over any whitespace characters at the beginning of `nptr`. It stops converting when it reaches a character that can't be part of a number and stores a pointer to the rest of the string in `endptr`.

`strtod()` looks for a number formatted like this:

`[+|-] [`*digits*`] [. `*digits*`] [e|E [-|+] `*digits*`]`

Anything in brackets (`[]`) is optional. If items are separated by vertical bars (`|`), only one of those items may be in that position.

See Also

`atof()`, `atoi()`, `atol()`, `sscanf()`, `strtol()`, `strtoul()`

Returns

`strtod()` returns these values:

If the number in nptr...	The function returns...
is valid	the double precision floating point value.
is outside the allowed range	HUGE_VAL, with the same sign as the number. It also sets `errno` to ERANGE.
causes underflow	0 (zero). It also sets `errno` to ERANGE.

Example

This program converts some strings to floating point numbers:

```
#include <stdio.h>
#include <stdlib.h>

main()
{
   char *s1 = "3.14159";
   char *s2 = "123.45E+676HIKE!";
   double result;
   char *end;

   result = strtod( s1, &end );
   printf( "s1 is %G, with \"%s\" "
           "left over.\n", result, end );

   result = strtod( s2, &end );
   printf( "s2 is %G, with \"%s\" "
           "left over.\n", result, end );
}
```

The program prints this:

```
s1 is 3.14159, with "" left over.
s2 is 1.2345E+678, with "HIKE!" left over.
```

strtok

Library

ANSI, ANSI-small, ANSI-A4

Syntax

```
#include <string.h>
char *strtok(char *s1, const char *s2);
```

Description

`strtok()` breaks the string `s1` into tokens, which are separated by one or more characters from the string of delimiters `s2`. It returns a pointer to the next token. To find all the tokens in a string, you must call `strtok()` in a loop, once for each token.

The first time you call `strtok()`, `s1` should be the string you want to break into tokens. `strtok()` saves this string. On each subsequent call, `s1` should be NULL. `strtok()` searches for the next token in the string it saved. `s2` can be different from call to call

If `strtok()` can't find a token, it returns NULL. If `strtok()` finds a token, it returns a pointer to the first character in it. If the token is not at the end of the string, `strtok()` replaces the delimiter with a null character (`'\0'`).

To reset `strtok()` and break another string into tokens, call `strtok()` again with the new string.

Note

Notice that `strtok()` changes the string `s1`. After each call, it replaces a delimiter with a null character (`'\0'`).

Returns

`strtok()` returns these values:

If the function...	**it returns...**
finds a token	a pointer to the token.
doesn't find a token	NULL.

Example

This program finds the words in a sentence and prints them out, one to a line.

```
#include <stdio.h>
#include <string.h>

main()
{
   /* The delimiters are a space,    */
   /* a comma, and a period.         */
   char *delims = " ,.";
   char *string = "Madam, I'm Adam.";
   char *token;

   /* The first time you call strtok(),       */
   /* include the string. strtok() saves it.  */
   token = strtok( string, delims );

   while ( token != NULL ) {

      printf( "%s\n", token );

      /* Each subsequent time you call        */
      /* strtok(), use NULL. strtok() uses    */
      /* the string it saved.                 */
      token = strtok( NULL, delims );
   }
}
```

The program prints out these tokens:

```
Madam
I'm
Adam
```

strtol

Library

ANSI, ANSI-small, ANSI-A4

Syntax

```
#include <stdlib.h>
long int strtol(const char *nptr, char **endptr,
                int base);
```

Description

`strtol()` converts a string to a long integer value. It converts the number represented in `nptr` to a long integer of base `base`, skipping over any whitespace characters in the beginning of the string. It stops converting when it reaches a character that can't be part of a number and stores a pointer to the rest of the string in `endptr`.

The argument `base` must be 0 (zero) or between 2 to 36. If `base` is between 2 and 36, `strtol()` converts a number in that base. If `base` is 0 (zero), `strtol()` examines the number's prefix to figure the base:

If the prefix is...	the base is...
0x or 0X	16 (hexadecimal)
0 (zero)	8 (octal)
none	10 (decimal)

`strtol()` looks for a number formatted like this:

```
[+|-][0x|0X|0][digits]
```

Anything in brackets ([]) is optional. If items are separated by vertical bars (|), only one of those items may be in that position.

Note

Hexadecimal numbers (base 16) may begin with 0x or 0X. The X will stop the conversion of other numbers for which X is not a valid digit.

digits must contain numerals and letters that are valid for the base `base`. For example, if `base` is 8, the number may contain only the numerals 0-7. If the base is 20, the number may contain only the numerals 0-9 and the letters A-J and a-j.

Returns

`strtol()` returns these values:

If the function...	The function returns...
finds a valid number	the long integer value.
doesn't find a number	0 (zero)
finds a number over LONG_MAX	LONG_MAX. It also sets errno to ERANGE.
finds a number under LONG_MIN	LONG_MIN. It also sets errno to ERANGE.

See Also

`atof()`, `atoi()`, `atol()`, `sscanf()`, `strtod()`, `strtoul()`

Example

This example converts two strings to long integer values:

```
#include <stdio.h>
#include <stdlib.h>

main()
{
   char *s1 = "982398247boo!";
   char *s2 = "-0xFACES";
   long int result;
   char *end;

   result = strtol( s1, &end, 10 );
   printf( "s1 is %ld, with \"%s\" "
           "left over.\n", result, end );

   result = strtol( s2, &end, 0 );
   printf( "s2 is %ld, with \"%s\" "
           "left over.\n", result, end );
}
```

The program prints this:

```
s1 is 982398247, with "boo!" left over.
s2 is -64206, with "S" left over.
```

strtoul

Library

ANSI, ANSI-small, ANSI-A4

Syntax

```
#include <stdlib.h>
unsigned long int strtoul(const char *nptr,
        char **endptr, int base);
```

Description

`strtoul()` converts a string to an unsigned long integer value. It converts the number represented in `nptr` to an unsigned long integer of base `base`, skipping over any whitespace characters in the beginning of the string. It stops converting when it reaches a character that can't be part of a number and stores a pointer to the rest of the string in `endptr`.

The argument `base` must be 0 or between 2 and 36. If `base` is between 2 and 36, `strtoul()` converts a number in that base. If `base` is 0 (zero), `strtoul()` examines the number's prefix to figure the base:

If the prefix is...	the base is...
0x or 0X	16 (hexadecimal)
0 (zero)	8 (octal)
none	10 (decimal)

`strtoul()` looks for a number formatted like this:

`[+][0x|0X|0][`*digits*`]`

Anything in brackets (`[]`) is optional. If items are separated by vertical bars (`|`), only one of those items may be in that position.

Note

Hexadecimal numbers (base 16) may begin with `0x` or `0X`. The `X` will stop the conversion of other numbers for which `X` is not a valid digit.

---

digits must contain numerals and letters that are valid for the base `base`. For example, if `base` is 8, the number may contain only the numerals `0-7`. If the base is 20, the number may contain only the numerals `0-9` and the letters `A-J` and `a-j`.

Returns

`strtoul()` returns these values:

If the function...	The function returns...
finds a valid number	the unsigned long integer value.
doesn't find a number	0 (zero)
finds a number over `ULONG_MAX`	`ULONG_MAX`. It also sets `errno` to `ERANGE`.

See Also

`atof()`, `atoi()`, `atol()`, `sscanf()`, `strtod()`, `strtol()`

Example

This example converts two strings to long integer values:

```
#include <stdio.h>
#include <stdlib.h>

main()
{
    char *s1 = "3723682357boo!";
    char *s2 = "0xFACES";
    unsigned long int result;
    char *end;

    result = strtoul( s1, &end, 10 );
    printf( "s1 is %lu, with \"%s\" "
            "left over.\n", result, end );

    result = strtoul( s2, &end, 0 );
    printf( "s2 is %lu, with \"%s\" "
            "left over.\n", result, end );
}
```

The program prints this:

```
s1 is 3723682357, with "boo!" left over.
s2 is 64206, with "S" left over.
```

strxfrm

Library ANSI

Syntax

```
#include <string.h>
size_t strxfrm(char *s1, const char *s2, size_t n);
```

Description

`strxfrm()` copies the first n characters of string `s2` into the string `s1`, without performing any transformation. If n is zero, `s1` can be `NULL`. `strxfrm()` is equivalent to

```
strncpy(s1, s2, n);
```

except that `strxfrm()` returns the length of `s2`.

`strxfrm()` is provided for ANSI C compatibility only. According to the standard, it transforms the string `s2` (preparing it for `strcmp()`) and copies it into the string `s1`.

To localize your software for other countries, use the International Utilities Package, described in *Inside Macintosh VI*, Chapter 14, "Worldwide Software Overview."

Returns The length of `s2`.

system

Library ANSI

Syntax

```
#include <stdlib.h>
int system(const char *string);
```

Description

`system()` ignores its argument and always returns 0 (zero)

`system()` is provided for ANSI C compatibility only. According to the standard, `system()` asks the computer to execute the command line in `string`.

Note

In a portable program, you can use `system(NULL)` to test for a command processor. If there is a command processor, `system(NULL)` returns a non-zero value. Otherwise, it returns zero.

To launch an application named in a string, use `exec()` on page 148.

Returns 0 (zero)

See Also `exec()`

tan

Library ANSI

Syntax

```
#include <math.h>
double tan(double x);
```

Description `tan()` returns the tangent of x, in radians.

Returns `tan()` returns these values:

If x is...	The function returns...
valid	the tangent of x.
an odd multiple of $\pi/2$	Infinity. It also sets `errno` to EDOM.

See Also `acos()`, `asin()`, `atan()`, `atan2()`, `cos()`, `sin()`, `tanh()`

Example This program prints the tangent of any number you enter:

```
#include <stdio.h>
#include <math.h>

main()
{
   double x, result;

   printf( "Enter an angle: " );
   scanf( "%lf", &x );

   result = tan(x);

   printf( "tan(%G) = %G\n", x, result);
}
```

This is output from the program:

```
Enter an angle: 1
tan(1) = 1.55741
```


tanh

Library ANSI

Syntax

```
#include <math.h>
double tanh(double x);
```

Description `tanh()` returns the hyperbolic tangent of x.

Returns The hyperbolic tangent of x.

See Also `atan()`, `cosh()`, `sinh()`, `tan()`

Example This program prints the hyperbolic tangent of any number you enter:

```
#include <stdio.h>
#include <math.h>

main()
{
   double x, result;

   printf( "Enter an angle: " );
   scanf( "%lf", &x );

   result = tanh(x);

   printf( "tanh(%G) = %G\n", x, result);
}
```

This is output from the program:

```
Enter an angle: 1
tanh(1) = 0.761594
```

tell

Not in the ANSI standard.

Library

unix

Syntax

```
#include <unix.h>
long tell(int fn);
```

Description

To learn how the standard libraries handle streams, see "Using Files and Streams" on page 15.

`tell()` returns the current file position for the file with file number `fn`. It is identical to `ftell()`, except that `tell()` takes a file number instead of a file descriptor.

For binary files, `tell()` returns the offset from the beginning of the file in bytes. For text files, its return value isn't necessarily useful, except as an argument to `lseek()`.

Returns

`tell()` returns these values:

If the function...	It returns...
succeeds	the value of the file position indicator
fails	EOF

See Also

`ftell()`, `fseek()`, `lseek()`

Example

This program opens a file for updating, saves your position, writes a string, restores your old position, and reads in the string.

```
#include <stdio.h>
#include <fcntl.h>
#include <unix.h>

main()
{
   int len, fn;
   long int pos;
   char *s1 = "Hello, world.\n";
   char *s2 = "How are you?\n";
   char buf[80];

   fn = creat( "NewFile", 0 );
   write( fn, s1, strlen(s1) );

   pos = tell( fn );

   write( fn, s2, strlen(s2) );
   lseek( fn, pos, SEEK_SET );
   len = read( fn, buf, 80 );
   printf( "You read: %.*s\n", len, buf );
   close( fn );
}
```


The program writes this statement:

```
You read: How are you?
```

time

Library ANSI

Syntax

```
#include <time.h>
time_t time(time_t *timer);
```

Description `time()` returns the current calendar time. It also assigns the time to `timer`, if `timer` is not NULL.

In THINK C, the current calendar time is the number of seconds since Midnight, January 1, 1904.

Returns The current calendar time.

See Also `asctime()`, `clock()`, `ctime()`, `difftime()`, `localtime()`, `mktime()`, `strftime()`

Example This prints the current calendar time:

```
#include <stdio.h>
#include <time.h>

main()
{
   time_t now;
   struct tm *date;
   char s[80];

   /* Get the current calendar time.    */
   time( &now );

   /* Convert it to a broken-down time.*/
   date = localtime( &now );

   /* Convert that to a string.         */
   strftime( s, 80, "%c", date );

   printf( "%s\n", s );
}
```

The program prints the time like this:

```
Fri Sep 21 17:49:14 1990
```


tmpfile

Library ANSI, ANSI-small

Syntax

```
#include <stdio.h>
FILE *tmpfile(void)
```

Description `tmpfile()` creates a temporary file. It opens a binary for updating (using `fopen()` with a mode of `"wb+"`).

To learn how the standard libraries handle streams, see "Using Files and Streams" on page 15.

Temporary files are automatically deleted when they're closed or when your program ends.

Returns `tmpfile()` returns these values:

If the function...	It returns...
creates the file	a pointer to the stream.
can't create the file	NULL.

See Also `fclose()`, `fopen()`, `remove()`, `tmpnam()`

Example This program creates a temporary file and gives you a chance to look for it.

```
#include <stdio.h>

main()
{
   FILE *fp;

   /* This creates a temporary file. */
   fp = tmpfile();

   printf( "If you're running MultiFinder,\n"
           "check for the file now.\n"
           "Press Return when you're back.\n" );
   gets( s );

   fclose( fp );

   printf( "Now the file's gone.\n" );
}
```

tmpnam

Library ANSI, ANSI-small

Syntax

```
#include <stdio.h>
char *tmpnam(char *s);
```

Description tmpnam() returns a unique file name that you can use for a temporary file. It can create up to TMP_MAX (9,999) file names.

If s isn't NULL, tmpnam() sets it to the file name. It must be a pointer to an array with at least L_tmpnam (20) characters.

Note

tmpnam() does not delete its file when it's closed. You must delete it yourself with a function like remove().

Returns A unique file name.

See Also fclose(), fopen(), remove(), tmpfile()

Example This program opens a file with a name generated by tmpnam():

```
#include <stdio.h>

main()
{
   FILE *fp;
   char name[L_tmpnam];

   /* This creates a unique file name. */
   tmpname( name );

   /* This opens a file.                */
   fp = fopen( name, "w" );

   printf( "If you're running MultiFinder,\n"
           "check for the file now.\n"
           "Press Return when you're back.\n" );
   gets( s );

   fclose( fp );

   /* This deletes the file.            */
   remove( name );

   printf( "Now the file's gone.\n" );
}
```


tolower

Library ANSI, ANSI-small, ANSI-A4

Syntax

```
#include <ctype.h>
int tolower(int c);
```

Description tolower() converts a letter from uppercase to lowercase. If c is an uppercase letter (A-Z), tolower() returns the lowercase version (a-z). Otherwise, tolower() returns c, unchanged.

Note

tolower() does not convert international characters (those with ASCII codes over 0x80), like Â or Ç. To convert international characters, use LowerText() or Transliterate() described in *Inside Macintosh VI*.

Returns tolower() returns these values:

If c is...	The function returns...
an uppercase letter	the lowercase character.
not an uppercase letter	c, unchanged.

See Also toupper(), islower(), isupper()

Example This program asks for a character and converts it to lowercase:

```
#include <stdio.h>
#include <ctype.h>

main()
{
  char uc, lc;

  printf( "Enter an uppercase character: " );
  uc = getchar();

  lc = tolower( uc );

  if ( uc != lc )
    printf( "Lowercase '%c' is '%c'.\n",
            uc, lc );
  else
    printf( "There is no lowercase for '%c'.\n",
            uc );
}
```

This is sample output from the program:

```
Enter an uppercase character: A
Lowercase 'A' is 'a'.

Enter an uppercase character: ?
There is no lowercase for '?'.
```


toupper

Library ANSI, ANSI-small, ANSI-A4

Syntax

```
#include <ctype.h>
int toupper(int c);
```

Description `toupper()` converts a letter from lowercase to uppercase. If c is a lowercase letter (a-z), `toupper()` returns the uppercase version (A-Z). Otherwise, `tolower()` returns c, unchanged.

Note

`toupper()` does not convert international characters (those with ASCII codes over 0x80), like *ä* or *ç*. To convert international characters, use `UpperText()` or `Transliterate()` described in *Inside Macintosh VI*.

Returns `toupper` returns these values:

If c is...	The function returns...
a lowercase letter	the uppercase character.
not a lowercase letter	c, unchanged.

See Also `tolower()`, `islower()`, `isupper()`

Example This program asks for a character and converts it to uppercase:

```
#include <stdio.h>
#include <ctype.h>

main()
{
    char uc, lc;

    printf( "Enter a lowercase character: " );
    lc = getchar();

    uc = toupper( lc );

   if ( uc != lc )
     printf( "Uppercase '%c' is '%c'.\n",
             lc, uc );
   else
     printf( "There is no uppercase for '%c'.\n",
             uc );
 }
```

This is sample output from the program:

```
Enter a lowercase character: a
Uppercase 'a' is 'A'.

Enter a lowercase character: =
There is no uppercase for '='.
```


ungetc

Library

ANSI, ANSI-small

Syntax

```
#include <stdio.h>
int ungetc(int c, FILE *stream);
```

Description

To learn how the standard libraries handle streams, see "Using Files and Streams" on page 15.

ungetc() pushes a single character c back onto the input stream stream. The next read from that stream will get that character first.

A stream can remember at least one pushed-back character. The maximum number it can remember depends on how big and how full its buffer is.

A successful intervening call to a file-positioning function (fseek(), fsetpos(), or rewind()) for the stream discards any pushed-back characters for that stream.

Make sure that c is between 0 and 255. ungetc() converts it to an unsigned char.

If c is EOF, ungetc() fails and the input stream is unchanged.

Returns

ungetc() returns these values:

If the function...	It returns...
succeeds	c.
fails	EOF.

See Also

fgetc(), fgets(), getc(), getchar(), gets(), strtok()

Example

This reads a character from the file `MyFile`, pushes it back, and reads a string from the file:

```
#include <stdio.h>
#include <console.h>

main()
{
   FILE *fp;
   char c, s[5];

   fp = fopen( "MyFile", "r" );

   c = getc( fp );
   printf( "Got a character: %c\n", c );

   ungetc( c, fp );
   printf( "Put it back.\n" );

   fgets( s, 5, fp );
   printf( "Read the string: %s\n", s );

   fclose( fp );
}
```

If the `MyFile` contains this line:

```
wxyz
```

the program will print this:

```
Got a character: w
Put it back.
Read the string: wxyz
```


unlink

Not in the ANSI standard.

Library

`unix`

Syntax

```
#include <unix.h>
int unlink(char *filename);
```

Description

To learn how to specify file names, see "Naming files" on page 18.

`unlink()` deletes the file `filename`. It is identical to the function `remove()`.

If the file is open, `unlink()` returns an error and leaves the file unchanged.

See Also

`remove()`

Returns

`unlink()` returns these values:

If the function...	It returns...
succeeds	0
fails	-1

Example

This program deletes the file `MyFile`:

```
#include <unix.h>

main()
{
   unlink( "MyFile" );
}
```

va_start
va_arg
va_end

Library

ANSI, ANSI-small, ANSI-A4

Syntax

```
#include <stdarg.h>
void va_start(va_list ap, parmN);
type va_arg(va_list ap, type);
void va_end(va_list ap);
```

Description

These macros let you declare a function that can have a different number of arguments from call to call. Variable argument lists are useful when you want your function to handle any number of arguments. For example, you might want a function that prints out any number of items (like `printf()`) or a function that sums up any number of integers.

Functions with variable argument lists have two types of arguments. The **named arguments** are those that are declared the usual way in the function header. Your function always knows the number and type of named arguments. The **unnamed arguments** are those you don't declare in the parameter list. You use "`...`" in the parameter list to stand for the unnamed arguments. Your function doesn't know how many unnamed arguments or their types.

For example, `fprintf()` has two named arguments, `stream` and `format`, and any number of unnamed arguments:

```
int fprintf( FILE* stream, char *format, ... );
```

To set up the function with a variable argument list, do the following:

1. Declare your function. Its parameter list must contain at least one named argument. `va_start()` uses it to find the beginning of the variable argument list.

2. Be sure to put in three periods "`...`" at the end of your parameter list to stand for the unnamed arguments. *Don't* use the Macintosh character Option-;. This lets THINK C know that your function can take more arguments than the ones named. In this example, *parmN* is the last named argument in your parameter list:

   ```
   function( parmN, ... )
   ```


3. Within your function declare a variable of type `va_list`. This variable will point to each unnamed argument in turn. This is how to declare the argument pointer `ap`:

   ```
   va_list ap;
   ```

To use a function's unnamed parameters, do the following:

1. To initialize the argument pointer, call `va_start()`. In this example, `ap` is the argument pointer and *parmN* is the last named argument in your parameter list:

   ```
   va_start(ap, parmN);
   ```

2. To get an unnamed argument, call `va_arg()`. Each call returns one unnamed argument and steps `ap` to the next. `va_arg()` uses the type name *type* to determine what type of argument to return and how big a step down the argument stack to take:

   ```
   va_arg(ap, type);
   ```

3. To finish, call `va_end()`. You should call `va_end()` once after processing all its arguments but before exiting.

   ```
   va_end(ap);
   ```

Here are some important things to keep in mind when you use variable argument lists.

- Do not declare *parmN* to be a `char`, an `unsigned char`, a `float`, or any other type that hasn't been promoted. Also don't declare it to be a `register` variable, a function, or an array.
- `va_arg()` does not know when it's reached the end of the variable argument list. You must keep track of it yourself. For example, you could have always have the user end the argument list with the same value.

See Also

`vfprintf()`, `vprintf()`, `vsprintf()`

Example

This program calls a function that sums a list of numbers of any length.

```
#include <stdio.h>
#include <stdarg.h>

long int sum( int arg1, . . . );

main()
{
   long int result;

   result = sum( 2, 4, 6, 8, 0 );
   printf( "The sum of 2, 4, 6, and 8 "
           "is %ld.\n", result );
}

/* This sums up a list of integers. */
/* The last integer must be zero.   */
long int sum( int arg1, . . . )
{
   va_list ap;
   int next;
   long int total;

   total = arg1;

   va_start( ap, arg1 );

   while (next=va_arg(ap,int))
      total += next;

   va_end( ap );

   return( total );
}
```

The program prints this:

```
The sum of 2, 4, 6, and 8 is 20.
```


vfprintf

Library

ANSI, ANSI-small

Syntax

```
#include <stdio.h>
int vfprintf(FILE *stream, const char *format,
             va_list arg);
```

Description

For more information on variable-length argument lists, see `va_arg()`, `va_end()`, *and* `va_start()` *on page 318*

`vfprintf()` is similar to `fprintf()`, except that `vfprintf()` takes a pointer to an argument list as a parameter. You'll find `vfprintf()` especially useful when you want to modify the behavior of `fprintf()`.

Before you call `vfprintf()`, you must initialize `arg` with `va_start()`. After `vfprintf()` returns, you should call `va_end()`.

For complete information on the format argument, read Chapter 4, "Using printf and scanf."

Note

If you open a stream in text mode, `vfprintf()` converts Line Feed characters to Returns. For more information, see the Section "Reading from and writing to text files" earlier in this chapter.

Returns

`vfprintf()` returns these values:

If the function...	It returns...
succeeds	the number of characters written.
fails	a negative number.

See Also

`fprintf()`, `va_arg()`, `va_end()`, `va_start()`, `vprintf()`, `vsprintf()`.

Example

This program writes a list of results to the file `MyFile`.

```
#include <stdio.h>
#include <stdarg.h>

FILE *FP;
int PrintResults( char *, . . . );

main()
{
   FP = fopen( "MyFile", "w" );
   PrintResults( "√16=%d   pi=%7.5f  x=%d",
                 4, 3.14159, 10 );
   fclose( FP );
}
```

```
int PrintResults( char *format, . . . )
{
   va_list ap;
   int result;

   fputs( "The results are as follows:\n", FP );

   va_start( ap, format );
   result = vfprintf( FP, format, ap );
   va_end( ap );

   return( result );
}
```

After you run the program, MyFile contains these lines:

```
The results are as follows:
√16=4   pi=3.14159  x=10
```


vprintf

Library

ANSI, ANSI-small

Syntax

```
#include <stdio.h>
int vprintf(const char *format, va_list arg);
```

Description

vprintf() is similar to printf(), except that vprintf() takes a pointer to an argument list as a parameter. You'll find vprintf() especially useful when you want to modify the behavior of printf().

Before you call vprintf(), you must initialize arg with va_start(). After vprintf() returns, you must deallocate arg with va_end().

Returns

vprintf() returns these values:

If the function...	It returns...
succeeds	the number of characters written.
fails	a negative number.

See Also

printf(), va_arg(), va_end(), va_start(), vfprintf(), vsprintf().

Example

For more information on variable-length argument lists, see va_arg(), va_end(), *and* va_start() *on page 318*

For complete information on the format argument, read Chapter 4, "Using printf and scanf."

This program prints a list of results:

```
#include <stdio.h>
#include <stdarg.h>

int PrintResults( char *, . . . );

main()
{
   PrintResults( "√16=%d   pi=%7.5f  x=%d",
                 4, 3.14159, 10 );
}

int PrintResults( char *format, . . . )
{
   va_list ap;
   int result;

   fputs( "The results are as follows:\n", FP );

   va_start( ap, format );
   result = vprintf( format, ap );
   va_end( ap );
   return( result );
}
```

The program writes these lines to standard output:

```
The results are as follows:
√16=4   pi=3.14159  x=10
```


vsprintf

Library

ANSI, ANSI-small, ANSI-A4

Syntax

```
#include <stdio.h>
int vsprintf(char *s, const char *format,
             va_list arg);
```

Description

`vsprintf()` is similar to `sprintf()`, except that `vsprintf()` takes a pointer to an argument list as a parameter. You'll find `vsprintf()` especially useful when you want to modify the behavior of `sprintf()`.

Before you call `vsprintf()`, you must initialize `arg` with `va_start()`. After `vsprintf()` returns, you must deallocate `arg` with `va_end()`.

Returns

`vsprintf()` returns these values:

If the function...	It returns...
succeeds	the number of characters written to the string.
fails	a negative number.

See Also

`sprintf()`, `va_arg()`, `va_end()`, `va_start()`, `vfprintf()`, `vprintf()`.

Example

This program writes a list of results to the file `MyFile`.

For more information on variable-length argument lists, see `va_arg()`, `va_end()`, *and* `va_start()` *on page 318*

For complete information on the format argument, read Appendix 4, "Using printf and scanf."

```
#include <stdio.h>
#include <stdarg.h>

char SuperString[512];
int FormatResults( char *, . . . );

main()
{
   FILE *fp;

   FormatResults( "√16=%d   pi=%7.5f  x=%d",
                  4, 3.14159, 10 );

   fp = fopen( "MyFile", "w" );
   fputs( SuperString, fp );
   fclose( fp );
}
```

```
int FormatResults( char *format, . . . )
{
   va_list ap;
   char s[128]="The results are as follows:\n";
   int result;

   strcat( s, format );

   va_start( ap, format );
   result = vfprintf( SuperString, s, ap );
   va_end( ap );

   return( result );
}
```

After you run the program, `MyFile` contains these lines:

```
The results are as follows:
√16=4   pi=3.14159  x=10
```


wctomb

Library

ANSI

Syntax

```
#include <stdlib.h>
int wctomb(char *s, wchar_t wchar);
```

Description

`wctomb()` sets the first element of `s` to `wchar`, if `s` is not NULL.

`wctomb()` is provided for ANSI C compatibility only. THINK C does not support multi-byte characters. According to the standard, `wctomb()` converts `wchar` to a multi-byte character, stores it in the array `s` points to, and returns the length in bytes of the multi-byte character.

To use characters that are more than one-byte long, use the Script Manager, described in *Inside Macintosh VI*, Chapter 14, "Worldwide Software Overview.

Returns

`wctomb()` returns these values:

If s is...	The function returns...
not NULL	1.
NULL	0.

wcstombs

Library ANSI

Syntax

```
#include <stdlib.h>
size_t wcstombs(wchar_t *pwcs, const char *s,
                size_t n);
```

Description

`wcstombs()` copies the elements of `pwcs` to `s`, stopping after it copies `n` elements or the null character (`'\0'`).

`wcstombs()` is provided for ANSI C compatibility only. THINK C does not support multi-byte characters. According to the standard, `wcstombs()` converts the sequence of codes `pwcs` to a multi-byte character string, stores it in the array `s` points to, and returns the number of array elements changed.

To use characters that are more than one-byte long, use the Script Manager, described in *Inside Macintosh VI*, Chapter 14, "Worldwide Software Overview

Returns The number of characters copied.

write

Not in the ANSI standard.

Library `unix`

Syntax

```
#include <unix.h>
int write(int fn, char *buffer, unsigned nbytes);
```

Description

To learn how the standard libraries handle streams, see "Using Files and Streams" on page 15.

`write()` copies bytes from a buffer to a file.

These are the arguments to `write()`:

Argument	Description
`fn`	File number of the file to write to.
`buffer`	Buffer to copy bytes from.
`nbytes`	Number of bytes to copy.

`write( fn, buffer, sizeof(Foo)*3 )`, where `buffer` is declared as `Foo buffer[3]`, is identical to

```
fwrite( buffer, sizeof(Foo), 3, fp );
```

except that `write()` takes a file number instead of a file descriptor.

Returns `write()` returns these values:

If the function...	It returns...
succeeds	the number of bytes copied
fails	`-1`

See Also `read()`, `open()`, `close()`.

Example This code fragment writes out 3 items of the structure `Foo` to the file `MyFooFile`, and then reads them back in.

```
#include <stdio.h>
#include <fcntl.h>
#include <unix.h>

typedef struct {
   int a;
   char b;
} Foo;

main ()
{
   int fn;
   int i;
   Foo inbuf[3];
```

writing
bytes 195, 243, 329
characters 73, 177, 239, 241
errors 236, 281
floating point numbers 63–69
formatted output 49–92, 175, 238, 321, 323
integers 55–59
Macintosh Toolbox types 75–77
strings 71–72, 178, 242
strings, Pascal 76–77
strings, to 271, 325
structures 195, 329
switching to reading 255
time, to string 104, 145, 282

X

%x conversion character 55, 57, 60

Z

zero flag 59, 68
zero padding 59

wcstombs

Library ANSI

Syntax

```
#include <stdlib.h>
size_t wcstombs(wchar_t *pwcs, const char *s,
                size_t n);
```

Description

`wcstombs()` copies the elements of `pwcs` to `s`, stopping after it copies `n` elements or the null character (`'\0'`).

`wcstombs()` is provided for ANSI C compatibility only. THINK C does not support multi-byte characters. According to the standard, `wcstombs()` converts the sequence of codes `pwcs` to a multi-byte character string, stores it in the array `s` points to, and returns the number of array elements changed.

To use characters that are more than one-byte long, use the Script Manager, described in *Inside Macintosh VI*, Chapter 14, "Worldwide Software Overview

Returns The number of characters copied.

write

Not in the ANSI standard.

Library unix

Syntax

```
#include <unix.h>
int write(int fn, char *buffer, unsigned nbytes);
```

Description

To learn how the standard libraries handle streams, see "Using Files and Streams" on page 15.

`write()` copies bytes from a buffer to a file.

These are the arguments to `write()`:

Argument	Description
`fn`	File number of the file to write to.
`buffer`	Buffer to copy bytes from.
`nbytes`	Number of bytes to copy.

`write( fn, buffer, sizeof(Foo)*3 )`, where `buffer` is declared as `Foo buffer[3]`, is identical to

```
fwrite( buffer, sizeof(Foo), 3, fp );
```

except that `write()` takes a file number instead of a file descriptor.

Returns `write()` returns these values:

If the function...	It returns...
succeeds	the number of bytes copied
fails	`-1`

See Also `read()`, `open()`, `close()`.

Example This code fragment writes out 3 items of the structure `Foo` to the file `MyFooFile`, and then reads them back in.

```
#include <stdio.h>
#include <fcntl.h>
#include <unix.h>

typedef struct {
   int a;
   char b;
} Foo;

main ()
{
   int fn;
   int i;
   Foo inbuf[3];
```

LDBL_MIN_EXP 24
ldexp() 212
ldiv() 213
left justifying 58, 67, 71, 73
length of string 284
limits.h file 24–26
line buffered, *See* buffering
line feeds, converted in files 20
lines, clearing 120
locale 36
localeconv() 214
localtime() 215
log files 122, 125
log() 216
log10() 217
logarithm, base ten 217
logarithm, natural 216
login name 203
long double, *See* floating point numbers
long integer, *See* integers
LONG_MAX 26
LONG_MIN 26
longjmp() 220
lowercase letters
  converting to 311
  testing for 208
lseek() 218

M

Macintosh Toolbox error codes 15
Macintosh Toolbox types, reading and writing 75–77
malloc() 222
maximum exponent 24
maximum floating point value 24
maximum integer values 24–26
maximum open streams 17
MAXSIZ 21
MB_LEN_MAX 25
mblen() 223
mbstowcs() 224
mbtowc() 225
MC68881, ANSI library for 13
memchr() 226
memcmp() 227
memcpy() 228
memmove() 229
memory 20–21
  allocating 119, 222, 251
  clearing 119
  comparing 227
  copying 228, 229
  freeing 181
  initializing 230
  reallocating 251
  searching in 226
  See also strings
memset() 230
minimum exponent 24
minimum floating point value 24
minimum integer values 24–26
- flag 58, 67, 71, 73
minus signs, writing 56, 67
mktime() 231
modf() 233
modulo 147, 170, 213
moving
  memory 228, 229
  strings 279, 289
moving cursor 130
moving file position 189, 191, 218
MS-DOS, porting programs 39
multibyte characters 35

N

%n conversion character 80
naming files 18–19, 254, 310
NaN, reading and writing 68
natural logarithm 216
newlines, converted in files 20
Not a Number, reading and writing 68
numbers, *See* digits, floating point numbers, integers
numbers,file 17

O

%o conversion character 55, 57, 60
octal numbers
  converting from strings 298, 300
  reading 60, 62
  writing 55, 57
  See also integers
open() 234
opening consoles 40, 44, 174
opening files 139, 171, 234, 309
OSType, reading and writing 75–76

P

%p conversion character 57
padding integers 58
parsing tokens 296
partial pathnames 19
Pascal strings, reading and writing 76–77
pathnames 18
pausing an application 41, 270
percent sign, reading and writing 71, 79
permErr 15
perror() 236
+ flag 56, 67
plus signs, writing 56, 67
pointers
  writing 56
  See also memory
porting programs 39
position of console 44
position, cursor 129, 130
positions, file
  getting 164, 193, 306
  setting 189, 191, 218
flag 57, 76
pow() 237
precision 85
  argument list, in 78–79
precision, floating point 23
"press <return> to exit" title 41
printf() 51, 238
printing
  console contents 122, 126
  files 42, 126
  See also writing
printing characters, testing for 208
process id 201
programs
  porting 39
  See also applications
punctuation, testing for 208
putc() 239
putchar() 241
puts() 242
putw() 243

Q

qsort() 244
quicksort 244
quitting 101, 149, 150
  See also exiting
quotient 147, 213

R

raise() 246
raising signals 41, 210, 246
raising to a power 154, 237
rand() 248
random numbers 248, 273
range error 14
read() 249
reading
  arrow keys 46
  bytes 179, 205, 249
  characters 75, 162, 197, 199
  command keys 46
  floating point numbers 69–71
  formatted input 49–92, 187, 256
  integers 60–63
  Macintosh Toolbox types 75–77
  Pascal strings 76–77
  pushing back character 315
  strings 73–74, 166, 204
  strings, from 274, 294, 298, 300
  structures 179, 249
  switching to writing 255
  tabs 46
realloc() 251
reallocating memory 251
redirecting I/O 122, 182, 184
remainder, from division 147, 170, 213
remove() 253
removing files 253, 317
rename() 254
renaming files 254
reopening streams 182, 184
ResType, reading and writing 75–76
return, pressing to exit 41
returns, converted in files 20
reverse video 46–47, 133
rewind() 255
rewinding files 255
right justifying 58, 67, 71, 73
rounding numbers 128, 169, 233

S

%#s conversion character 76
%s conversion character 71, 73
SANE, ANSI library for 13
scan sets 82–83
scanf() 52, 256
SCHAR_MAX 25
SCHAR_MIN 25
screen, clearing 121
  See also consoles
searching 116
  for character in string 276, 291
  for set of characters in string 280, 290, 292

C_BREAK 45–46
C_ECHO 45–46
C_NOECHO 45–46
C_RAW 45–46
calendar time 21
calloc() 119
canceling operations 41
ccleol() 120
ccleos() 121
ccommand() 122
cecho2file() 125
cecho2printer() 126
ceil() 128
ceiling 128
cgetxy() 129
cgotoxy() 130
char, *See* characters, integers
CHAR_BIT 25
CHAR_MAX 26
CHAR_MIN 25
characters 27, 30
  classifying 208
  lowercase 311
  multibyte 35
  pushing back onto stream 315
  reading 75, 162, 197, 199
  searching for 226, 276, 291
  testing 208
  uppercase 313
  wide 35
  writing 73, 177, 239, 241
chide() 131
Chooser user name 203
cinverse() 133
clearerr() 134
clearing consoles 120, 121
clearing errors 134
clearing memory 119
clock time 21, 135
clock() 135
clock_t type 21
close() 136
closing files 136, 156
command keys, reading 46
command line, emulating 122
Command-Period, handling 41–43
comparing
  memory 227
  strings 277, 287
compiling ANSI library 12
concatenating strings 275, 285
conditions, testing 106
console functions, ANSI library without 11, 13
console interface 40–44
console.h file 44
console_options structure 44
consoles 28, 37–47
  blocking input 46
  buffering 45–46, 140
  clearing 120, 121
  closing 156
  creating 40, 44, 174
  echoing characters 45–46
  editing in 45
  file, echoing to 125
  font in 44
  hiding and showing 131, 143
  input modes 45–46, 140
  inverse video 46–47, 133
  outside console interface 44
  position of 44
  position, cursor 129, 130
  printer, echoing to 126
  reopening 184
  size of 44
  tabs 142
  testing if console 207
  title of 44
  See also streams
control characters, testing for 208
conversion characters 87–89, 91–92
conversions
  characters, case 311, 313
  files, UNIX and ANSI 157, 168
  string from anything 271, 325
  string to anything 274
  string to floating point 113, 294
  string to integer 114, 115, 298, 300
  time to string 104, 145, 282
  time, broken-down and calendar 215, 231
  time, local to UTC 206
copying
  memory 228, 229
  strings 279, 289
cos() 137
cosh() 138
cosine 137
creat() 139
creating ANSI library 12
creating consoles 40, 44, 174
creating files 139, 171, 234, 309
csetmode() 140
csettabs() 142
cshow() 143
ctime() 145
current time 308
cursor position 129, 130

D

%d conversion character 55, 60
dates, *See* time
day of week 232
DBL_DIG 23, 24
DBL_EPSILON 24
DBL_MANT_DIG 23, 24
DBL_MAX 24
DBL_MAX_10_EXP 24
DBL_MAX_EXP 24
DBL_MIN 24
DBL_MIN_10_EXP 24
DBL_MIN_EXP 24
default folder 18
deleting files 253, 317
descriptors, file 17
diagnostics 106
difftime() 146
digits, testing for 208
direct I/O 179, 195, 205, 243, 249, 329
directories, *See* folders
dirNFErr 15
div() 147
division 147, 170, 213
domain error 14
DOS, porting programs 39
double, *See* floating point numbers
dskFulErr 15
dupFNErr 15
dynamically allocating memory 20–21, 119, 222

E

%e conversion character 65, 69
e, raising to a power 154
EACCES 15
EBADF 14
EBUSY 15
echoing
  to console 45–46
  to file 125
  to printer 126
editing input 45
EDOM 14
EEXIST 15
EINTR 14
EINVAL 14
EIO 15
EMFILE 14, 17
end of file testing 158
end-of-file character 45
ENFILE 15, 17
ENODEV 14
ENOENT 15
ENOSPC 15
ENOTDIR 15
environment 202, 220, 260
EOF character 45
EOF testing 158
ERANGE 14
EROFS 15
errno variable 13–15
errors 13–15
  checking for, in streams 159
  clearing, in streams 134
  diagnosing 106
  end of file 158
  Macintosh Toolbox codes 15
  messages, error 236, 281
  streams and files 14
ESRCH 14
exec() 148
execl() 148
execle() 148
execlp() 148
execlpe() 148
executing program 148
execv() 148
execve() 148
execvp() 148
execvpe() 148
_exit() 150
exit() 149
exiting 101, 149, 150
  functions running at 109, 111
  launching application and 148
  on Command-Period 41
  on condition 106
  pausing before 41
  preparing for 152
  pressing return before 41
_exiting() 152
exp() 154
exponent 154, 237
exponent limits, floating point 24

C_BREAK 45–46
C_ECHO 45–46
C_NOECHO 45–46
C_RAW 45–46
calendar time 21
calloc() 119
canceling operations 41
ccleol() 120
ccleos() 121
ccommand() 122
cecho2file() 125
cecho2printer() 126
ceil() 128
ceiling 128
cgetxy() 129
cgotoxy() 130
char, *See* characters, integers
CHAR_BIT 25
CHAR_MAX 26
CHAR_MIN 25
characters 27, 30
  classifying 208
  lowercase 311
  multibyte 35
  pushing back onto stream 315
  reading 75, 162, 197, 199
  searching for 226, 276, 291
  testing 208
  uppercase 313
  wide 35
  writing 73, 177, 239, 241
chide() 131
Chooser user name 203
cinverse() 133
clearerr() 134
clearing consoles 120, 121
clearing errors 134
clearing memory 119
clock time 21, 135
clock() 135
clock_t type 21
close() 136
closing files 136, 156
command keys, reading 46
command line, emulating 122
Command-Period, handling 41–43
comparing
  memory 227
  strings 277, 287
compiling ANSI library 12
concatenating strings 275, 285
conditions, testing 106
console functions, ANSI library without 11, 13
console interface 40–44
console.h file 44
console_options structure 44
consoles 28, 37–47
  blocking input 46
  buffering 45–46, 140
  clearing 120, 121
  closing 156
  creating 40, 44, 174
  echoing characters 45–46
  editing in 45
  file, echoing to 125
  font in 44
  hiding and showing 131, 143
  input modes 45–46, 140
  inverse video 46–47, 133
  outside console interface 44
  position of 44
  position, cursor 129, 130
  printer, echoing to 126
  reopening 184
  size of 44
  tabs 142
  testing if console 207
  title of 44
  See also streams
control characters, testing for 208
conversion characters 87–89, 91–92
conversions
  characters, case 311, 313
  files, UNIX and ANSI 157, 168
  string from anything 271, 325
  string to anything 274
  string to floating point 113, 294
  string to integer 114, 115, 298, 300
  time to string 104, 145, 282
  time, broken-down and calendar 215, 231
  time, local to UTC 206
copying
  memory 228, 229
  strings 279, 289
cos() 137
cosh() 138
cosine 137
creat() 139
creating ANSI library 12
creating consoles 40, 44, 174
creating files 139, 171, 234, 309
csetmode() 140

csettabs() 142
cshow() 143
ctime() 145
current time 308
cursor position 129, 130

D

%d conversion character 55, 60
dates, *See* time
day of week 232
DBL_DIG 23, 24
DBL_EPSILON 24
DBL_MANT_DIG 23, 24
DBL_MAX 24
DBL_MAX_10_EXP 24
DBL_MAX_EXP 24
DBL_MIN 24
DBL_MIN_10_EXP 24
DBL_MIN_EXP 24
default folder 18
deleting files 253, 317
descriptors, file 17
diagnostics 106
difftime() 146
digits, testing for 208
direct I/O 179, 195, 205, 243, 249, 329
directories, *See* folders
dirNFErr 15
div() 147
division 147, 170, 213
domain error 14
DOS, porting programs 39
double, *See* floating point numbers
dskFulErr 15
dupFNErr 15
dynamically allocating memory 20–21, 119, 222

E

%e conversion character 65, 69
e, raising to a power 154
EACCES 15
EBADF 14
EBUSY 15
echoing
  to console 45–46
  to file 125
  to printer 126
editing input 45
EDOM 14
EEXIST 15
EINTR 14
EINVAL 14
EIO 15
EMFILE 14, 17
end of file testing 158
end-of-file character 45
ENFILE 15, 17
ENODEV 14
ENOENT 15
ENOSPC 15
ENOTDIR 15
environment 202, 220, 260
EOF character 45
EOF testing 158
ERANGE 14
EROFS 15
errno variable 13–15
errors 13–15
  checking for, in streams 159
  clearing, in streams 134
  diagnosing 106
  end of file 158
  Macintosh Toolbox codes 15
  messages, error 236, 281
  streams and files 14
ESRCH 14
exec() 148
execl() 148
execle() 148
execlp() 148
execlpe() 148
executing program 148
execv() 148
execve() 148
execvp() 148
execvpe() 148
_exit() 150
exit() 149
exiting 101, 149, 150
  functions running at 109, 111
  launching application and 148
  on Command-Period 41
  on condition 106
  pausing before 41
  preparing for 152
  pressing return before 41
_exiting() 152
exp() 154
exponent 154, 237
exponent limits, floating point 24

LDBL_MIN_EXP 24
ldexp() 212
ldiv() 213
left justifying 58, 67, 71, 73
length of string 284
limits.h file 24–26
line buffered, *See* buffering
line feeds, converted in files 20
lines, clearing 120
locale 36
localeconv() 214
localtime() 215
log files 122, 125
log() 216
log10() 217
logarithm, base ten 217
logarithm, natural 216
login name 203
long double, *See* floating point numbers
long integer, *See* integers
LONG_MAX 26
LONG_MIN 26
longjmp() 220
lowercase letters
  converting to 311
  testing for 208
lseek() 218

M

Macintosh Toolbox error codes 15
Macintosh Toolbox types, reading and writing 75–77
malloc() 222
maximum exponent 24
maximum floating point value 24
maximum integer values 24–26
maximum open streams 17
MAXSIZ 21
MB_LEN_MAX 25
mblen() 223
mbstowcs() 224
mbtowc() 225
MC68881, ANSI library for 13
memchr() 226
memcmp() 227
memcpy() 228
memmove() 229
memory 20–21
  allocating 119, 222, 251
  clearing 119
  comparing 227
  copying 228, 229
  freeing 181
  initializing 230
  reallocating 251
  searching in 226
  See also strings
memset() 230
minimum exponent 24
minimum floating point value 24
minimum integer values 24–26
- flag 58, 67, 71, 73
minus signs, writing 56, 67
mktime() 231
modf() 233
modulo 147, 170, 213
moving
  memory 228, 229
  strings 279, 289
moving cursor 130
moving file position 189, 191, 218
MS-DOS, porting programs 39
multibyte characters 35

N

%n conversion character 80
naming files 18–19, 254, 310
NaN, reading and writing 68
natural logarithm 216
newlines, converted in files 20
Not a Number, reading and writing 68
numbers, *See* digits, floating point numbers, integers
numbers,file 17

O

%o conversion character 55, 57, 60
octal numbers
  converting from strings 298, 300
  reading 60, 62
  writing 55, 57
  See also integers
open() 234
opening consoles 40, 44, 174
opening files 139, 171, 234, 309
OSType, reading and writing 75–76

P

%p conversion character 57
padding integers 58
parsing tokens 296
partial pathnames 19
Pascal strings, reading and writing 76–77
pathnames 18
pausing an application 41, 270
percent sign, reading and writing 71, 79
permErr 15
perror() 236
+ flag 56, 67
plus signs, writing 56, 67
pointers
  writing 56
  See also memory
porting programs 39
position of console 44
position, cursor 129, 130
positions, file
  getting 164, 193, 306
  setting 189, 191, 218
flag 57, 76
pow() 237
precision 85
  argument list, in 78–79
precision, floating point 23
"press <return> to exit" title 41
printf() 51, 238
printing
  console contents 122, 126
  files 42, 126
  See also writing
printing characters, testing for 208
process id 201
programs
  porting 39
  See also applications
punctuation, testing for 208
putc() 239
putchar() 241
puts() 242
putw() 243

Q

qsort() 244
quicksort 244
quitting 101, 149, 150
  See also exiting
quotient 147, 213

R

raise() 246
raising signals 41, 210, 246
raising to a power 154, 237
rand() 248
random numbers 248, 273
range error 14
read() 249
reading
  arrow keys 46
  bytes 179, 205, 249
  characters 75, 162, 197, 199
  command keys 46
  floating point numbers 69–71
  formatted input 49–92, 187, 256
  integers 60–63
  Macintosh Toolbox types 75–77
  Pascal strings 76–77
  pushing back character 315
  strings 73–74, 166, 204
  strings, from 274, 294, 298, 300
  structures 179, 249
  switching to writing 255
  tabs 46
realloc() 251
reallocating memory 251
redirecting I/O 122, 182, 184
remainder, from division 147, 170, 213
remove() 253
removing files 253, 317
rename() 254
renaming files 254
reopening streams 182, 184
ResType, reading and writing 75–76
return, pressing to exit 41
returns, converted in files 20
reverse video 46–47, 133
rewind() 255
rewinding files 255
right justifying 58, 67, 71, 73
rounding numbers 128, 169, 233

S

%#s conversion character 76
%s conversion character 71, 73
SANE, ANSI library for 13
scan sets 82–83
scanf() 52, 256
SCHAR_MAX 25
SCHAR_MIN 25
screen, clearing 121
  See also consoles
searching 116
  for character in string 276, 291
  for set of characters in string 280, 290, 292

wcstombs

Library ANSI

Syntax

```
#include <stdlib.h>
size_t wcstombs(wchar_t *pwcs, const char *s,
                size_t n);
```

Description

`wcstombs()` copies the elements of `pwcs` to `s`, stopping after it copies `n` elements or the null character (`'\0'`).

`wcstombs()` is provided for ANSI C compatibility only. THINK C does not support multi-byte characters. According to the standard, `wcstombs()` converts the sequence of codes `pwcs` to a multi-byte character string, stores it in the array `s` points to, and returns the number of array elements changed.

To use characters that are more than one-byte long, use the Script Manager, described in *Inside Macintosh VI*, Chapter 14, "Worldwide Software Overview"

Returns The number of characters copied.

write

Not in the ANSI standard.

Library unix

Syntax

```
#include <unix.h>
int write(int fn, char *buffer, unsigned nbytes);
```

Description

To learn how the standard libraries handle streams, see "Using Files and Streams" on page 15.

`write()` copies bytes from a buffer to a file.

These are the arguments to `write()`:

Argument	Description
`fn`	File number of the file to write to.
`buffer`	Buffer to copy bytes from.
`nbytes`	Number of bytes to copy.

`write( fn, buffer, sizeof(Foo)*3 )`, where `buffer` is declared as `Foo buffer[3]`, is identical to

```
fwrite( buffer, sizeof(Foo), 3, fp );
```

except that `write()` takes a file number instead of a file descriptor.

Returns `write()` returns these values:

If the function...	It returns...
succeeds	the number of bytes copied
fails	`-1`

See Also `read()`, `open()`, `close()`.

Example This code fragment writes out 3 items of the structure `Foo` to the file `MyFooFile`, and then reads them back in.

```
#include <stdio.h>
#include <fcntl.h>
#include <unix.h>

typedef struct {
   int a;
   char b;
} Foo;

main ()
{
   int fn;
   int i;
   Foo inbuf[3];
```

writing
bytes 195, 243, 329
characters 73, 177, 239, 241
errors 236, 281
floating point numbers 63–69
formatted output 49–92, 175, 238, 321, 323
integers 55–59
Macintosh Toolbox types 75–77
strings 71–72, 178, 242
strings, Pascal 76–77
strings, to 271, 325
structures 195, 329
switching to reading 255
time, to string 104, 145, 282

X

%x conversion character 55, 57, 60

Z

zero flag 59, 68
zero padding 59